**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**
**„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”**

# ОБЧИСЛЮВАЛЬНА ГЕОМЕТРІЯ В ЗАДАЧАХ КОМП’ЮТЕРНОЇ ГРАФІКИ ТА КОМП’ЮТЕРНОГО ЗОРУ

Конспект лекцій

Харків 2018

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**
**„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”**

# ОБЧИСЛЮВАЛЬНА ГЕОМЕТРІЯ В ЗАДАЧАХ КОМП’ЮТЕРНОЇ ГРАФІКИ ТА КОМП’ЮТЕРНОГО ЗОРУ

Конспект лекцій для студентів спеціальності 122 Комп’ютерні науки

Затверджено
редакційно-видавничою
радою університету,
протокол № 3 від 10.10.18 р.

Харків
НТУ «ХПІ»
2018

Обчислювальна геометрія в задачах комп’ютерної графіки та комп’ютерного зору. Конспект лекцій для студентів спеціальності 122 Комп’ютерні науки / Уклад. Дашкевич А.О. – Харків : НТУ «ХПІ», 2018. – 46 с.

Укладач А. О. Дашкевич

Рецензент Д. В. Воронцова

Кафедра геометричного моделювання та комп'ютерної графіки

**ЗМІСТ**

## ВСТУП

Обчислювальна геометрія – галузь комп’ютерних наук, що вивчає способи побудови ефективних алгоритмів для розв'язання геометричних задач. Об’єктами вивчення обчислювальної геометрії є дискретні геометричні об’єкти, базовими з яких є точка та відрізок прямої, на основі яких будуються складні об’єкти, такі як багатокутники (полігони) в двовимірному просторі та їх узагальнення в тривимірному просторі – багатогранники (поліедри).

Сучасна підготовка студентів зі спеціальності 122 Комп’ютерні науки потребує викладання студентам матеріалу з методів обчислювальної геометрії, які використовуються в галузях комп'ютерної графіки та комп'ютерного зору. Конспект лекцій, який запропоновано автором, відповідає зазначеній потребі.

Структурно конспект складається:

- зі змісту;
- вступу;
- викладення теоретичного матеріалу з 8 лекцій;
- списку джерел інформації.

# 1 ВИЗНАЧЕННЯ ОБЧИСЛЮВАЛЬНОЇ ГЕОМЕТРІЇ. ОСНОВНІ ОБ'ЄКТИ І МЕТОДИ ОБЧИСЛЮВАЛЬНОЇ ГЕОМЕТРІЇ. ПОНЯТТЯ СКЛАДНОСТІ ОБЧИСЛЕНЬ. КЛАСИ СКЛАДНОСТІ. RAM–МАШИНА

*Обчислювальна геометрія* – галузь комп'ютерних наук, що вивчає способи побудови ефективних алгоритмів для розв'язання геометричних задач. Об'єктами вивчення обчислювальної геометрії є дискретні геометричні об'єкти, базовими з яких є *точка* та *відрізок прямої*, на основі яких будуються складні об'єкти, такі як *багатокутники* (полігони) в двовимірному просторі та їх узагальнення в тривимірному просторі – *багатогранники* (поліедри).

Основними методами обчислювальної геометрії є:

1. Замітання.
2. Розділяй і володарюй.
3. Відсічення та пошук.
4. Зведення задач.

*Складність обчислень* – функція залежності об'єму роботи, яка виконується деяким алгоритмом, від розміру вхідних даних. Об'єм роботи зазвичай вимірюється як кількість необхідних часу та простору – *обчислювальні ресурси*. Час визначається як кількість елементарних кроків алгоритму для вирішення задачі, в той час як простір визначається об'ємом пам'яті, необхідним для роботи алгоритму. Основним питанням теорії складності обчислень є визначення зміни часу виконання алгоритму і об'єму пам'яті в залежності від розміру вхідних даних.

*Часова складність алгоритму* – функція від розміру вхідних даних, що дорівнює максимальній кількості елементарних операцій, які виконуються алгоритмом для розв'язання задачі встановленого розміру. *Просторова складність* визначається аналогічним образом для об'єму пам'яті, що використовується під час роботи алгоритму.

Кількість і час виконання елементарних операцій алгоритму залежить не тільки від кількості даних, але й від структури самих даних, тому, для абстрагування від конкретної технічної платформи для роботи алгоритму, обчислювальну складність виражають з допомогою *асимптотичної складності* – це член розкладання функції складності, який зростає найшвидше зі збільшенням вхідних даних n, всі члени меншого порядку при цьому ігноруються. Наприклад, якщо в процесі роботи алгоритму виконуються чотири

етапи, які мають часову складність $n^2$, $n \log n$, $n^3$ та $\log n$ відповідно, то розкладання функції складності матиме наступний вигляд:

$n^2 + n \log n + n^3 + \log n = n^3$ (після відкидання членів меншого порядку).

Асимптотична складність визначає порядок складності незалежно ні від точного часу виконання окремих операцій, ні від кількості бітів пам'яті, яку займають змінні, ні від швидкості процесора, на якому виконується алгоритм. Для позначення асимптотичної складності використовують спеціальну нотацію "О велике" (табл. 1.1).

Таблиця 1.1 – Асимптотична складність алгоритмів

| Позначення | Пояснення | Визначення |
|---|---|---|
| $f(n) \in O(g(n))$ | Функція $f$ обмежена зверху функцією $g$ з точністю до постійного множника $C$ | $\|f(n)\| \le \|C \cdot g(n)\|$ |
| $f(n) \in \Omega(g(n))$ | Функція $f$ обмежена знизу функцією $g$ з точністю до постійного множника $C$ | $\|C \cdot g(n)\| \le \|f(n)\|$ |
| $f(n) \in \Theta(g(n))$ | Функція $f$ обмежена зверху і знизу функцією $g$ асимптотично | $\|C \cdot g(n)\| \le \|f(n)\| \le \|C' \cdot g(n)\|$ |

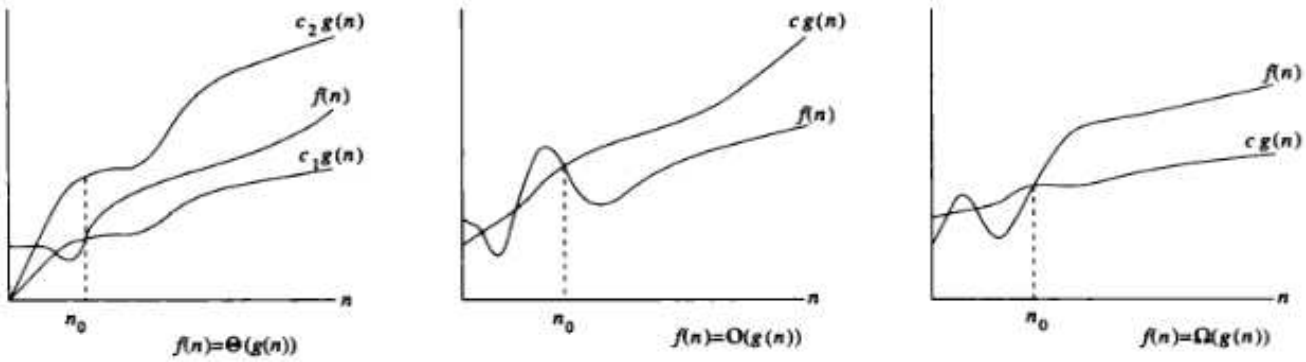


Рис. 1.1. О–нотація

Таким чином, складність $O(n^2)$ вказує на те, що часова складність алгоритму квадратична від кількості вхідних даних.

Виділяють поняття складності в *найгіршому випадку* та складності в *середньому* (*очікувана* складність).

Поширені види часової складності алгоритмів наведено в табл. 1.2.

Таблиця 1.2 – Приклади часової складності алгоритмів

| Час роботи | Пояснення, приклад часу виконання | Приклад |
|---|---|---|
| $O(1)$ | Постійний час виконання | Визначення парності числа в двійковому запису |
| $O(\log^* n)$ | Повторний логарифмічний час | |
| $O(\log \log n)$ | Подвійний логарифмічний час | |
| $O(\log n)$ | Логарифмічний час | Двійковий пошук у впорядкованому масиві |
| $O(\text{poly}(\log n))$ | Полілогарифмічний час, $\log^2 n$ | |
| $O(n^C)$, $0<C<1$ | Дрібний ступень | Пошук в *kd*–дереві |
| $O(n)$ | Лінійний час | Пошук максимуму в масиві, пошук елемента у невпорядкованому масиві |
| $O(n \log^* n)$ | | Триангуляція Зейделя |
| $O(n \log n)$ | Лінійно–логарифмічний час | Сортування злиттям та порівнянням |
| $O(n^2)$ | Квадратичний час | Бульбашкове сортування |
| $O(n^3)$ | Кубічний час | Множення матриць |
| $O(\text{poly}(n))$ | Поліноміальний час, $n$, $n^2$, $n \log n$, $n^{10}$ | |
| $O(2^{\text{poly}(\log n)})$ | Квазіполіноміальний час, $n^{\log n}$ | |
| $O(\ 2^{n^\epsilon}, \epsilon \geq 0\ )$ | Підекспоненційний час, $2^{\log n^{\log \log n}}$ | |
| $O(c^{\text{poly}(n)})$ | Експоненційний час, $2^n$ | Алгоритми пошуку повним перебором |
| $O(n!)$ | Факторіальний час | Комбінаторні алгоритми: пошук усіх перестановок або комбінацій |
| $O(\ 2^{2^{poly(n)}}\ )$ | Подвійний експоненційний час | Доведення вірності твердження в арифметиці Пресбургера |

Основні класи задач обчислювальної геометрії:

1. Пошук підмножини: в таких задачах задається множина об'єктів і необхідно знайти деяку підмножину, яка задовільняє заданій умові. Наприклад, пошук найближчої пари на множині з $N$ точок або пошук вершин опуклої оболонки.
2. Задачі на обчислення: задається множина об'єктів, необхідно визначити величину деякого геометричного параметра на цій множині.
3. Задача розпізнавання: множина задач, які ставляться таким чином, що відповідь на задачу може бути тільки ТАК чи НІ. Наприклад, для заданої множини та деякого параметра цієї множини А, задача розпізнавання може бути представлена в вигляді питання "чи вірно, що для заданої множини параметр $A \geq A_0$". Задачі розпізнавання є задачами тісно пов'язаними з задачами пошуку підмножини та задач обчислення.

*Клас складності* – множина задач розпізнавання, для вирішення яких існують алгоритми, що схожі за обчислювальною складністю. Задачі розбиваються на класи згідно зі складністю їх розв'язання. Класи складності знаходяться в ієрархічному відношенні: одні класи містять у собі інші. Для більшості включень невідомо, чи є вони строгими. Найважливішими класами є такі:

- *клас P* – містить всі задачі, які можливо розв'язати за поліноміальний час на детермінованій машині Тюрінга, задачі з цього класу вважаються швидкими для розв'язання;
- *клас NP* – містить всі задачі, які можливо розв'язати за поліноміальний час на недетермінованій машині Тюрінга або перевірити за поліноміальний час на детермінованій машині Тюрінга, задачі з цього класу вважаються складними для розв'язання. Клас $P$ входить до класу NP.

Одним з найважливіших відкритих питань в галузі теоретичної інформатики є питання про рівність цих двох класів.

Функція $f(n)$ є *верхньою межею (оцінкою)* часової складності проблеми $P$, якщо існує алгоритм $A$, що розв'язує $P$ таким чином, що час роботи алгоритму $A$ складає $f(n)$. Функція $f(n)$ є *нижньою межею (оцінкою)* часової складності проблеми $P$, якщо будь–який алгоритм, що розв'язує $P$, має часову складність не меншу за $f(n)$.

*Перетворення (зведення) задач*. Для багатьох задач встановлення нижньої оцінки часу виконання – дуже складна задача, але для деяких задач можна

оцінити час виконання однієї задачі за відомим часом виконання іншої задачі методом перетворення (зведення) задач. Метод складається з наступних кроків:

1. Вихідні дані для задачі *A* перетворюються у відповідні вихідні дані для задачі *B*.
2. Розв'язується задача *B*.
3. Результат розв'язання задачі *B* перетворюється у вірне рішення задачі *A*.

В такому випадку задача *A* є такою, що зводиться до задачі *B*.

Оцінку складності алгоритмів проводять використовуючи поняття абстрактної обчислювальної машини. Найчастіше використовують модель *RAM–машини*, або машини з рівним доступом, яка складається з:

1. Вхідної стрічки, з якої проводиться зчитування чисел. Вхідна стрічка складається з послідовності комірок, в яких записані числа. При зчитуванні числа з комірки, головка машини переміщується вправо.
2. Вихідної стрічки, на яку проводиться запис чисел. Вихідна стрічка складається з послідовності комірок, які спочатку порожні. Під час запису в комірку, на яку вказує головка, зберігається число і головка машини переміщується вправо. Числа, що вже записані в комірки, не можна змінювати.
3. Пам'яті, яка складається з послідовності регістрів $r_0$, ..., $r_i$, ..., $r_N$, кожен з яких може зберігати довільне число. В регістрі $r_0$ проводяться обчислення, він має назву *сумaтор*.

Числа для RAM–машини можуть бути як цілими, так і дійсними.

Програма для RAM–машини не зберігається в її пам'яті і складається з послідовності команд.

Для RAM–машин наступні операції є елементарними:

1. Арифметичні операції та математичні функції.
2. Операції порівняння.
3. Операції непрямого доступу до пам'яті за деякою адресою.

Вказані операції виконуються за константний час.

# 2 ГЕОМЕТРИЧНІ ЗАСАДИ ОБЧИСЛЮВАЛЬНОЇ ГЕОМЕТРІЇ. БАГАТОКУТНИКИ. ЛОКАЛІЗАЦІЯ ТОЧКИ В БАГАТОКУТНИКУ. МЕТОД ЗАМІТАННЯ. ПОШУК ПЕРЕТИНІВ ВІДРІЗКІВ

*Простий багатокутник* $P$ – це замкнена ділянка площини, обмежена кінцевим набором відрізків прямих – *ребер*, що формують замкнену ламану без самоперетинів (окрім *зіркових* багатокутників). Загальна точка двох суміжних ребер – *вершина* багатокутника (рис. 2.1).

Множина вершин та ребер багатокутника формує його *границю* $\partial P$.

*Опуклий* багатокутник – багатокутник $P$, всі точки якого лежать на одній стороні від будь–якої прямої, що проходить через дві сусідні вершини $P$.

*Зірковий* багатокутник – багатокутник $P$, який має рівні сторони та рівні кути, та може мати самоперетини.

*Монотонний* багатокутник відносно деякої прямої $L$ – багатокутник $P$, границя якого має не більше двох точок перетину з прямою, що перпендикулярна $L$.

*Діагональ* багатокутника – відрізок, що зв'язує дві вершини $P$ та знаходиться у внутрішній частини $P$, не торкаючись його границі за винятком його кінцевих точок. Будь–який багатокутник з кількістю вершин більшою за три має діагональ. Діагоналі *неперетинні*, якщо вони не мають загальних внутрішніх точок.

*Теорема Жордана для багатокутників 2.1* Границя $\partial P$ багатокутника $P$ поділяє площину на дві частини – обмежену внутрішню (включає $\partial P$) і необмежену зовнішню таким чином, що будь–яка крива, що поєднує точку внутрішнього регіону з точкою зовнішнього регіону, повинна перетинати $\partial P$.

Рис. 2.1. Багатокутники: опуклий, зірковий

*Опорна пряма* до опуклого багатокутника $P$ – така пряма $L$, що проходить через деяку вершину $P$ таким чином, що усі внутрішні точки $P$ лежать на одній стороні від $L$.

*Евклідів простір* $E^k$ *розмірності* $k$ ($k$–мірний) – це простір усіх послідовностей (точок) з $k$ дійсних чисел $p = (x_1, ..., x_i, ..., x_k)$, $1 \le i \le k$. *Відстань* між двома точками $p_1=(x_1, ..., x_k)$ та $p_2 = (y_1, ..., y_k)$ в $k$–мірному Евклідовому просторі:

$$d(p_1, p_2)=\sum_{i=1}^{k}\left|y_i-x_i^2\right|^{1/2} \quad .$$

Пряма, що проходить через точки $p_1$ та $p_2$ може бути параметризована:

$$\alpha p_1+(1-\alpha)p_2 \quad ,$$

де α може приймати будь–яке дійсне значення. Якщо α обмежена інтервалом [0, 1], таке представлення є параметризацією відрізка $\overline{p_1p_2}$ , в якому точки $p_1$ та $p_2$ будуть його кінцями.

Узагальнюючи, $k$+1 незалежних точок $p_0$, $p_1$, ... $p_k$ належать до $k$–мірної гіперплощини, яка може бути параметризована:

$$\alpha_0 p_0+\alpha_1 p_1+...+\alpha_k p_k \quad ,$$

де $\sum_{i=0}^{k}\alpha_i=1$ , якщо обмежити усі $\alpha_i \ge 0$ , то така гіперплощина представляє симплекс на $k$+1 точках.

Якщо $T$ є трикутником на площині $E^2$ з вершинами $p_1=(x_1, y_1)$, $p_2=(x_2, y_2)$, $p_3=(x_3, y_3)$, то його подвійна *орієнтована (знакова)* площа (або *псевдоскалярний добуток векторів* $\overline{p_1p_2}$ та $\overline{p_2p_3}$ ):

$$2S=\left|(x_1 y_2-y_1 x_2)+(x_2 y_3-y_2 x_3)+(x_3 y_1-y_3 x_1)\right| \quad ,$$

або $2S=\left|(x_2-x_1)(y_3-y_2)-(y_2-y_1)(x_3-x_2)\right|$ .

де знак $S$ є позитивним, якщо ($p_1$, $p_2$, $p_3$) формують порядок обходу проти годинникової стрілки і негативним – якщо порядок обходу за годинниковою стрілкою.

*Пряма* $L$ на площині може бути представлена лінійним рівнянням:

$$Ax+By+C=0 \quad ,$$

таким чином, що точка $p = (x, y)$ знаходиться на прямій тоді і тільки тоді, якщо координати точки задовільняють рівнянню.

*Напівплощина*, що задана прямою $L$ може бути представлена або:

$$Ax+By+C\ge 0 \quad ,$$

або

$$Ax+By+C\le 0 \quad .$$

*Локалізація точки у опуклому багатокутнику.*

Заданий опуклий багатокутник $P$ на площині та деяка точка $Q$, необхідно визначити, чи знаходиться $Q$ всередині $P$ (рис. 2.2).

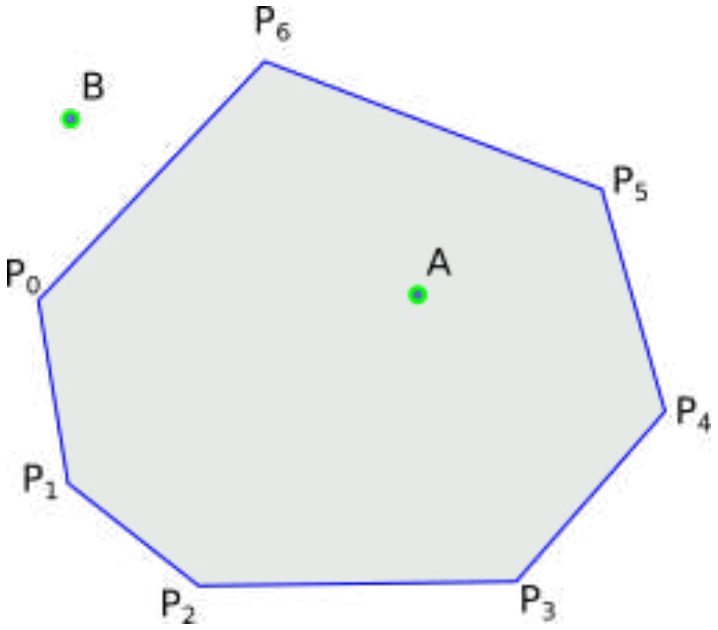

Рис. 2.2. Локалізація точки в багатокутнику

Наївний підхід: обійти усі ребра багатокутника проти (або за) годинниковою стрілкою, та визначити відносне положення $Q$ до кожного ребра:

- якщо $Q$ знаходиться по одну сторону від усіх ребер (зліва або справа), то $Q$ знаходиться всередині $P$, наприклад, на рис. 2.2 точка $A$ знаходиться по ліву сторону від усіх ребер при обході проти годинникової стрілки;
- якщо є хоча б одне ребро, для якого $Q$ знаходиться по іншу сторону, аніж відносно будь–якого іншого ребра, то $Q$ знаходиться за межами $P$, наприклад, на рис. 2.2 точка $B$ знаходиться по ліву сторону від ребер $P_0P_1$, $P_1P_2$, $P_2P_3$, $P_3P_4$, $P_4P_5$, $P_5P_6$, якщо здійснювати обхід проти годинникової стрілки, але по праву від ребра $P_6P_0$.

Часова складність такого алгоритму O($n$), де $n$ – кількість ребер багатокутника.

Існує алгоритм локалізації точки з часовою складністю O(log $n$):

1) Визначимо, чи знаходиться $Q$ всередині сегменту $P_{n-1}P_0P_1$, якщо ні, то точка знаходиться за межами $P$, завершуємо роботу алгоритму (рис. 2.3а);
2) Задаємо два індекси $p = 1$, $r = n - 1$ (рис. 2.3б);
3) Визначаємо індекс середньої вершини q = (p + r) / 2 (рис. 2.3б);
4) Якщо $Q$ зліва від $P_0P_q$, то $p = q$, якщо справа, то $r = q$ (рис. 2.3в);
5) Повторюємо кроки 3) та 4) поки $r - p > 1$ (рис. 2.3в–д);
6) Перевіряємо, чи перетинаються відрізки $P_0Q$ та $P_pP_r$, якщо перетинаються, то $Q$ лежить поза $P$, якщо ні, то $Q$ всередині $P$ (рис. 2.3д).

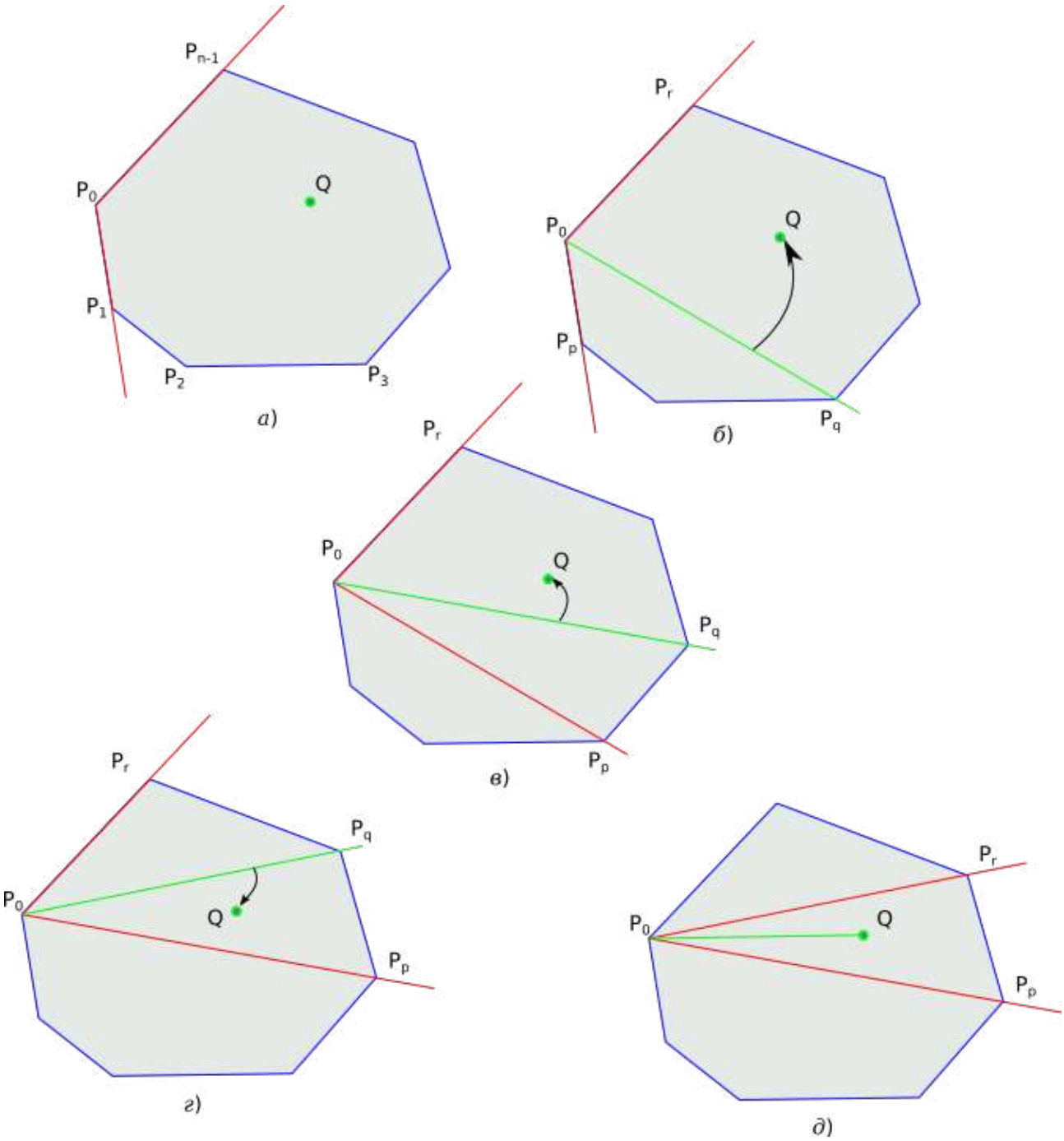


Рис. 2.3. Локалізація точки за O(log *n*)

*Пошук перетинів відрізків.*

Постановка проблеми знаходження перетинів: для заданих *n* відрізків на площині, знайти усі можливі перетини.

*Алгоритм Бентлі–Оттманна (метод замітання)*: задається вертикальна пряма *L*, яка переміщується (замітає) по площині зліва–направо. Для прямої задається змінна *STATUS*, що характеризує її поточний стан, в кожен момент часу *STATUS* містить усі відрізки, що перетинають пряму *L*, відсортовані за

y–координатою їх точок перетину з *L*. Стан замітаючої прямої змінюється в наступних випадках:

1. Пряма *L* доторкається лівого кінця відрізка *S*. В цьому випадку відрізок *S* не було розглянуто раніше і тому він має перетини з іншими відрізками з правої сторони прямої *L* і повинен бути доданий до змінної стану.
2. Пряма *L* доторкається правого кінця відрізка *S*. В цьому випадку відрізок *S* не може мати перетинів з іншими відрізками з правої сторони прямої *L* і повинен бути видалений зі змінної стану.
3. Пряма L доторкається точки перетина відрізків $S_1$ та $S_2$. В такому випадку відносні положення відрізків $S_1$ та $S_2$ повинні бути змінено на протилежні, так як відрізки в змінній стану відсортовані за *y*–координатою їх точок перетину з *L*.

Таким чином можна побачити, що статус прямої *L* не змінюється в процесі її руху зліва–направо до тих пір, поки не настане одна з можливих подій.

Також в алгоритмі вводиться структури *EVENT*, що міститиме множину точок–подій, до яких належать кінцеві точки відрізків та точки перетину відрізків, та *INTERSECTIONS*, в якій зберігатимуться точки перетинів. Точки–події сортуватимуться за їх x–координатами.

На рис. 2.4 показано схематичний процес роботи алгоритму:

- Зеленим кольором показані точки–події, що знаходяться в *EVENT* та відрізки, що знаходяться в *STATUS*;
- Червоним кольором показані точки, що вже видалені з *EVENT* та відрізки, що видалені з *STATUS*;
- Синім кольором показані відрізки, які ще не були розглянуті;
- Блакитним кольором показані точки–події, які є точками перетину відрізків.

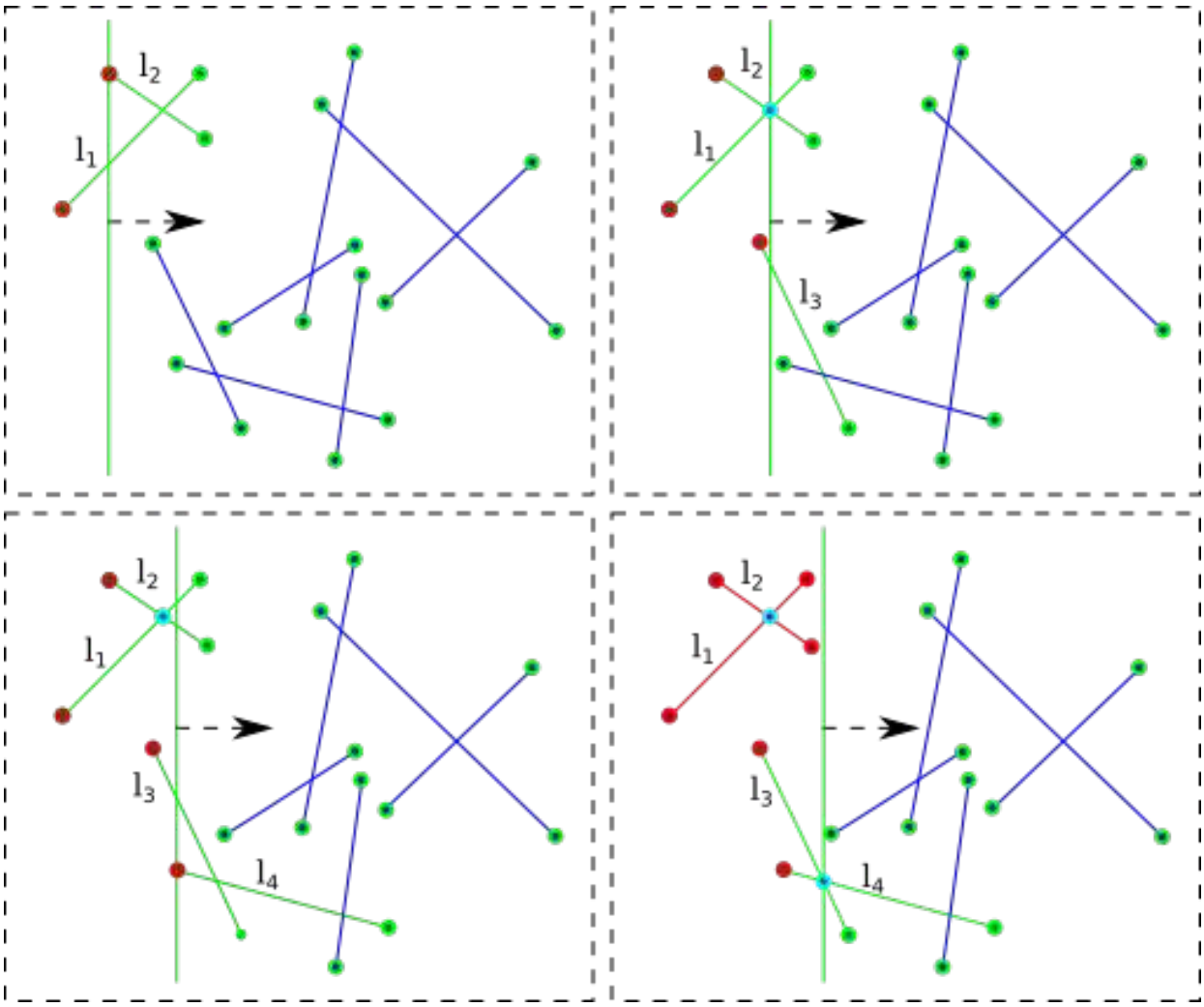

Рис. 2.4 Пошук перетинів відрізків методом замітання

Проаналізуємо алгоритм з точки зору часової складності. Сортування $2n$ точок на кроці (1) може бути виконано за час $O(n \log n)$, якщо використовувати ефективний алгоритм сортування, наприклад, сортування злиттям. Крок (2) займає константний час $O(1)$. Для розрахунку часу виконання циклу WHILE припустимо, що є $m$ точок перетину для заданих $n$ відрізків. Тоді маємо $n+m$ точок–подій. Якщо припустити, що операції пошуку мінімуму, вставки та видалення елемента можуть бути виконані кожна за час $O(\log n)$, то обробка кожного відрізка займатиме якнайбільше $O(\log n)$ часу та обробка кожної точки–події займатиме якнайбільше $O(\log(n+m))$ часу. Тоді алгоритм буде виконано за час:

$$O(n \log n) + O(1) + n \cdot O(\log n) + (n+m) \cdot O(\log(n+m)) = O((n+m)\log(n+m))$$

Можна відмітити, що в найгіршому випадку $m = n^2$, тоді $\log(n+m) = O(\log n)$ і часова складність складатиме:

$$O((n+m)\log n).$$

Таким чином часова складність залежить від кількості $m$ точок перетинів відрізків і в деяких випадках робота алгоритму не є ефективною. Наприклад, коли $m$ має порядок $\Omega(n^2)$, то алгоритм виконуватиметься за час $O(n^2 \log n)$, що навіть гірше аніж прямий метод, в якому береться кожна пара відрізків і обчислюється їх точка перетину. З іншого боку, коли число $m$ має порядок $\Omega(n)$, то алгоритм виконуватиметься ефективно за час $O(n \log n)$.

Лістинг 2.1. Алгоритм пошуку перетинів відрізків

```
Вхід: n відрізків S1, S2, ..., Sn
Вихід: усі точки перетину цих відрізків в INTERSECTIONS
1) Відсортуємо кінцеві точки відрізків і помістимо їх в EVENT
2) STATUS = ∅ (пуста множина)
INTERSECTIONS = ∅
3) WHILE EVENT ≠ ∅ (не пуста множина)
        p = MINIMUM(EVENT)
        ВИДАЛИТИ p з EVENT
        IF p є правим кінцем деякого відрізка S
                LET Si та Sj два відрізки, що суміжні з S в STATUS
                IF p є точкою перетину S з Si або Sj
                        ДОДАТИ p в INTERSECTIONS
                ВИДАЛИТИ S з STATUS
                IF Si та Sj перетинаються в точці p1 та x(p1) ≥ x(p)
                        ДОДАТИ p1 в EVENT
        ELSE IF p є лівим кінцем деякого відрізка S
                ДОДАТИ S в STATUS
                LET Si та Sj два відрізки, що суміжні з S в STATUS
                IF p є точкою перетину S з Si або Sj
                        ДОДАТИ p в INTERSECTIONS
                IF S перетинає Si в p1
                        ДОДАТИ p1 в EVENT
                IF S перетинає Sj в p2
                        ДОДАТИ p2 в EVENT
        ELSE IF p є точкою перетину Si та Sj та Si зліва від Sj в STATUS
                ДОДАТИ p в INTERSECTIONS
                змінити положення Si та Sj на протилежні
                LET Sk відрізок зліва від Sj та Sh відрізок справа від Si в STATUS
                IF Sk та Sj перетинаються в точці p1 та x(p1) > x(p)
                        ДОДАТИ p1 в EVENT
                IF Sh та Si перетинаються в точці p2 та x(p2) > x(p)
                        ДОДАТИ p2 в EVENT
```

# 3 ОПУКЛІ ОБОЛОНКИ

*Опукла оболонка CH(S)* множини *S* – перетин усіх опуклих множин, що містять *S* (рис. 3.1). Аналогічно, *CH(S)* – найменша опукла множина, що містить *S*.

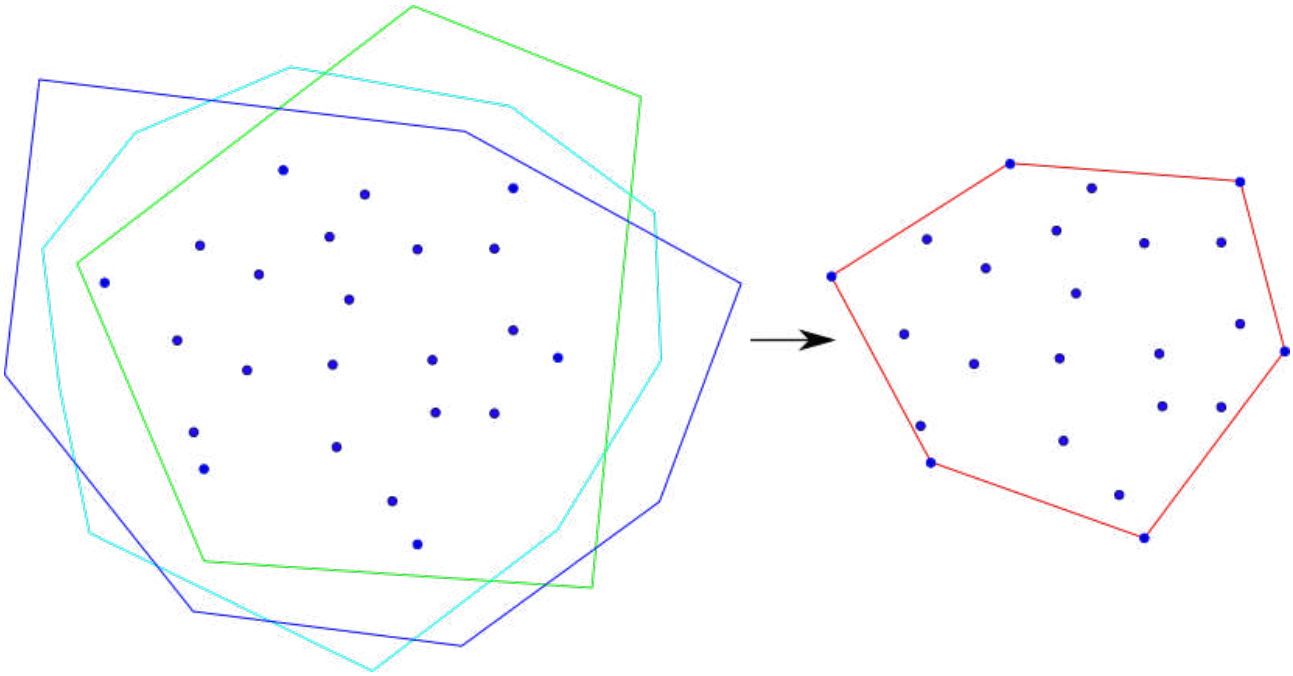

Рис. 3.1. Опукла оболонка множини точок

*Алгоритм Джарвіса (алгоритм загортання подарунку).*

Лістинг 3.1 Алгоритм Джарвіса

```
Вхід: множина S з n точками на площині
Вихід: опукла оболонка CH(S) множини S
1) LET p(0) – найлівіша точка в S з найменшою y–координатою
LET p(1) – точка в S така, що нахил відрізка p(0)p(1) до вісі x є найменшим
2) i = 1
3) WHILE p(i) ≠ p(0)
        FOR для кожної точки j від 0 до |S|, за винятком точок вже доданих до CH(S), але
                                                                включаючи p(0)
                LET p(i+1) – точка така, що кут <p(i–1)p(i)p(j) максимальний
        ДОДАТИ p(i+1) до CH(S)
        i = i + 1
```

Слід зазначити, що в алгоритмі необов'язково розраховувати значення кута $<p(i-1)p(i)p(j)$, достатньо визначити подвійну орієнтовану площу

трикутника $p(i-1)p(i)p(j)$ (псевдоскалярний добуток векторів $p(i-1)p(i)$ та $p(i)p(j)$).

Кроки (1) та (2) алгоритму будуть виконуватись за час $O(n)$ кожен. Крок (3) буде виконуватись за час $O(h \cdot n)$, де $h$ – кількість точок з $S$, що належать до опуклої оболонки. Таким чином повний час роботи алгоритму складатиме $O(h \cdot n)$. В найгіршому випадку, коли усі $n$ точок $S$ належать до $CH(S)$, час виконання складатиме $O(n^2)$. Схематичний процес роботи алгоритму показано на рис. 3.2.

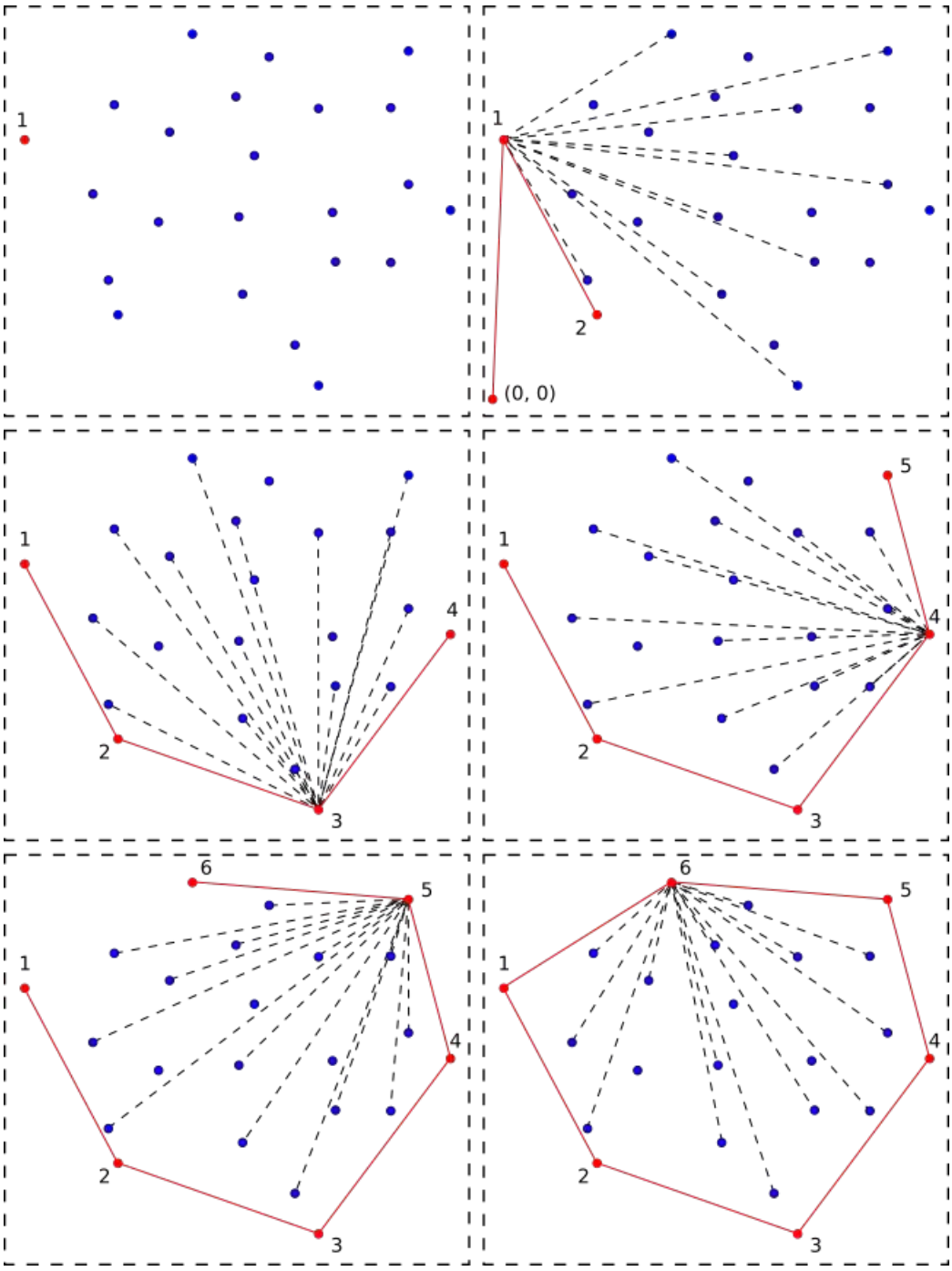


Рис. 3.2. Алгоритм Джарвіса

*Сканування Грехема*.

Лістинг 3.2 Алгоритм сканування Грехема

```
Вхід: множина S з n точками на площині
Вихід: опукла оболонка CH(S) множини S, яка знаходитиметься в стеку St
1) LET p(0) –точка в S з найменшою y–координатою (найлівіша, якщо таких точок
                                                                    декілька)
2) Відсортуємо точки в S за їх полярним кутом відносно p(0), якщо значення кута для
декількох точок співпадають, то за відстанню до p(0)
Отриманий відсортований список точок
L = { p(1), p(2), ..., p(n–1) }
3) PUSH(St, p(0))
PUSH(St, p(1))
4) i = 2
5) WHILE i ≤ n
        LET q(0) – точка на вершині стеку
        LET q(1) – точка стеку, наступна за q(0)
        IF точки q(1)q(0)p(i) не формують лівий поворот (лежать на одній прямій, або
                                                        формують правий поворот)
              POP(St)
        PUSH(St, p(i))
        i = i + 1
```

Розглянемо часову складність алгоритму. Крок (1) може бути виконаний за час O($n$), крок (2) – за час O($n$ log $n$), кроки (3) та (4) – за час O(1) кожен. Операції PUSH та POP на кроці (5) виконуються тільки по одному разу для кожної точки, так як точка, що була відкинута на більш ранній стадії алгоритму, більше не розглядається. Таким чином, на кроці (5) буде виконано 2$n$ операцій зі стеком і загальна складність кроку складатиме O($n$). Отримуємо загальну часову складність алгоритму Грехема O($n$ log $n$).

*Алгоритм Кіркпатрика методом "розділяй і володарюй"*. Загальна схема алгоритмів типу "розділяй і володарюй":

Лістинг 3.3. Алгоритм типу "розділяй і володарюй"

```
Вхід: задача P розміру n
Вихід: розв'язок P
1) IF n ≤ N0 (N0 – деяке невелике ціле число)
        розв'язуємо задачу P прямим методом та завершуємо роботу алгоритму
2) Розбиваємо P на k підзадач розміру n/k
3) Рекурсивно розв'язуємо кожну підзадачу
4) Об'єднуємо розв'язки підзадач для отримання розв'язку P
```

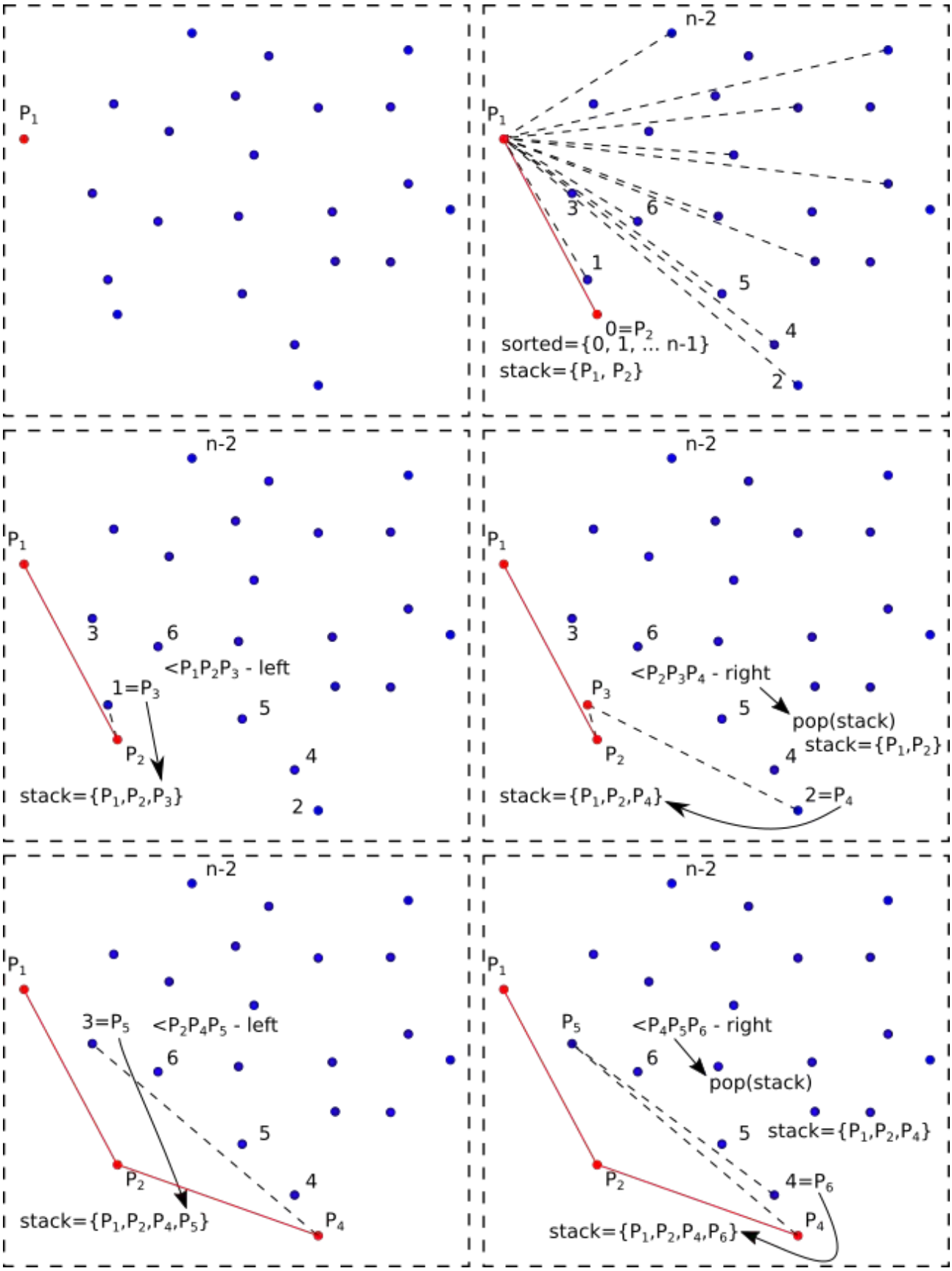


Рис. 3.3. Сканування Грехема

Лістинг 3.4. Алгоритм Кіркпатрика

```
Вхід: множина S з n точками на площині
Вихід: опукла оболонка CH(S) множини S
1) Відсортуємо точки в S за x−координатами
2) CALL MERGEHULL(S)

Функція MERGEHULL(S)
Вхід: множина S з n точками на площині, відсортованими за x−координатами
Вихід: опукла оболонка CH(S) множини S
1) IF S містить менш ніж чотири точки
        будуємо CH(S) прямим методом
ELSE
        розбиваємо S на дві підмножини S1 та S2 приблизно рівного розміру таким чином, що
        координата x будь−якої точки в S1 менше за x координату будь−якої точки в S2
2) CALL MERGEHULL(S1) для побудови опуклої оболонки CH(S1)
CALL MERGEHULL(S2) для побудови опуклої оболонки CH(S2)
3) CALL MERGE(CH(S1), CH(S2)) для отримання CH(S)

Функція MERGE(CH(S1), CH(S2))
Вхід: опуклі оболонки CH(S1) та CH(S2)
Вихід: об'єднана опукла оболонка CH(S1+S2) множин S1 та S2
1) Знаходимо верхню опорну пряму для CH(S1) та CH(S2)
CALL UPPER_BRIDGE(CH(S1), CH(S2))
2) Знаходимо нижню опорну пряму для CH(S1) та CH(S2)
CALL LOWER_BRIDGE(CH(S1), CH(S2))
3) LET u(S1) та l(S1) вершини в множині S1, що знаходяться на верхній та нижній опорних прямих, відповідно
LET u(S2) та l(S2) вершини в множині S2, що знаходяться на верхній та нижній опорних прямих, відповідно
Тоді усі вершини в CH(S1) в порядку обходу за годинниковою стрілкою від u(S1) до l(S1) можуть бути відкинуті
Аналогічно, усі вершини в CH(S2) в порядку обходу проти годинникової стрілки від u(S2) до l(S2) можуть бути відкинуті також
Усі точки в S1 та S2, що залишились, належать до об'єднаної опуклої оболонки CH(S1+S2)

Функція UPPER_BRIDGE(CH(S1), CH(S2))
Вхід: опуклі оболонки CH(S1) та CH(S2)
Вихід: верхня опорна пряма для CH(S1) та CH(S2)
1) LET p – точка в CH(S1) з найменшою x−координатою
LET q – точка в CH(S2) з найбільшою x−координатою
LET L – пряма через p та q
```

2) WHILE L не є верхньою опорною прямою
WHILE існує точка p’ сусідня до p в CH(S1), вища за пряму L
Замінюємо точку p на p’ та формуємо нову пряму L
WHILE існує точка q’ сусідня до q в CH(S2), вища за пряму L
Замінюємо точку q на q’ та формуємо нову пряму L
Функція LOWER_BRIDGE(CH(S1), CH(S2)) є симетричною до UPPER_BRIDGE(CH(S1), CH(S2)) і задається аналогічним чином, але знаходяться точки p’ та q’ в CH(S1) та в CH(S1), що знаходяться нижче за пряму L

Часова складність алгоритму Кіркпатрика складатиме O($n$ log $n$).

*Евристика Екла–Туссена* дозволяє видалити значну частину точок, що не належать до опуклої оболонки множини: знаходимо найлівішу, найправішу, найнижчу та найвищу точки множини $S$ (ці операції виконуються за час O($n$) кожна), вони формують опуклий чотирикутник $Q$, всі точки множини $S$, що знаходяться всередині $Q$ можуть бути відкинуті від подальшого розгляду, знаходження таких точок також виконується за час O($n$). Таким чином сумарна складність евристики O($n$).

Додаткові алгоритми побудови опуклих оболонок:

- *алгоритм Кіркпатрика–Зейделя методом “розділяй і володарюй”*, складність O($n$ log $h$);
- *алгоритм Чена*, складність O($nh$).

# 4 ТРІАНГУЛЯЦІЯ БАГАТОКУТНИКІВ

*Тріангуляція* багатокутника $P$ – це розбиття $P$ на трикутники максимальною множиною неперетинних діагоналей (рис. 4.1).

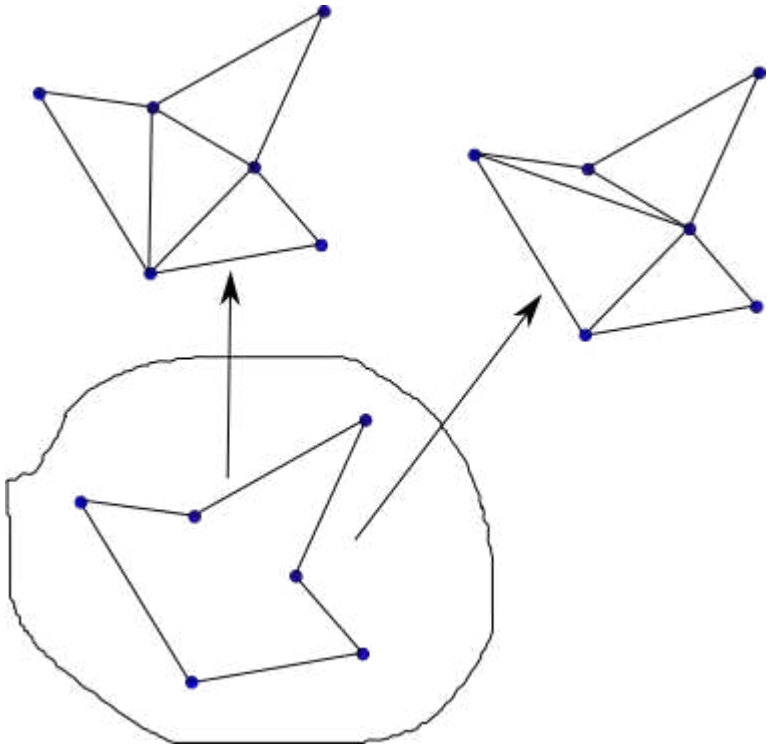

Рис. 4.1. Приклади триангуляції

*Теорема 4.1* Кожен простий багатокутник з $n$ точок має тріангуляцію з $n-2$ трикутників незалежно від тріангуляції.

*Примітивний алгоритм тріангуляції*: в загальному випадку в довільному багатокутнику усього $n^2$ варіантів побудови діагоналей. Для того, щоб перевірити всі варіанти, необхідно:

- визначити, чи перетинає діагональ багатокутник за $O(n)$ по усіх ребрах;
- визначити, чи належить діагональ внутрішній області багатокутника.

Усього необхідно знайти $n-3$ діагоналей. Сумарна часова складність алгоритму $O(n^4)$. Для опуклих багатокутників достатньо обрати одну з вершин та поєднати з іншими, за винятком суміжних з цією вершиною. В такому випадку часова складність зменшується до $O(n)$.

*Триангуляція монотонних багатокутників*. Сутність методу в тому, щоб розбити багатокутник на монотонні частини і тріангулювати кожну з них.

Простий багатокутник є *монотонним* відносно прямої $\ell$, якщо будь-яка пряма $\ell' \perp \ell$ перетинає його лише в двох точках. Багатокутник, монотонний відносно вісі $y$ має назву $y$-монотонний (рис. 4.2).

Лістинг 4.1. Алгоритм триангуляції монотонного багатокутника

Вхід: монотонний багатокутник P
Вихід: триангуляція P
1) Відсортуємо вершини в P за зменшенням y–координати
LET відсортований список L = {v(1), v(2), ..., v(n)}
2) Заносимо вершини v(1) та v(2) в стек S
3) LET i = 3
4) LET S = {u(1), u(2), ..., u(s)},
де u(s) – вершина стеку
5) IF v(i) суміжна до u(1), а не до u(s) (в цьому випадку усі вершини в стеку u(2), ..., u(s) видимі з v(i))
    додати ребра {v(i), u(2)}, {v(i), u(3)}, ..., {v(i), u(s)}
    POP усі вершини з стеку
    PUSH(u(s), S)
    PUSH(v(i), S)
    i++
    GOTO Крок 8
6) IF v(i) суміжна до u(s), а не до u(1) (в цьому випадку u(s) не видима з v(i))
    WHILE наступна за вершиною стека вершина u’ видима з v(i)
        Додати ребро {v(i), u’}
        POP вершину стеку u(s)
    PUSH(v(i), S)
    i++
    GOTO Крок 8
7) IF v(i) суміжна як до u(1), так і до u(s) (в цьому випадку v(i) є останньою в L і усі вершини в стеку, за винятком u(1) та u(s) видимі з v(1)
    додати ребра {v(i), u(2)}, {v(i), u(3)}, ..., {v(i), u(s–1)}
    POP усі вершини з стеку та STOP
8) IF i <= n, GOTO Крок 4

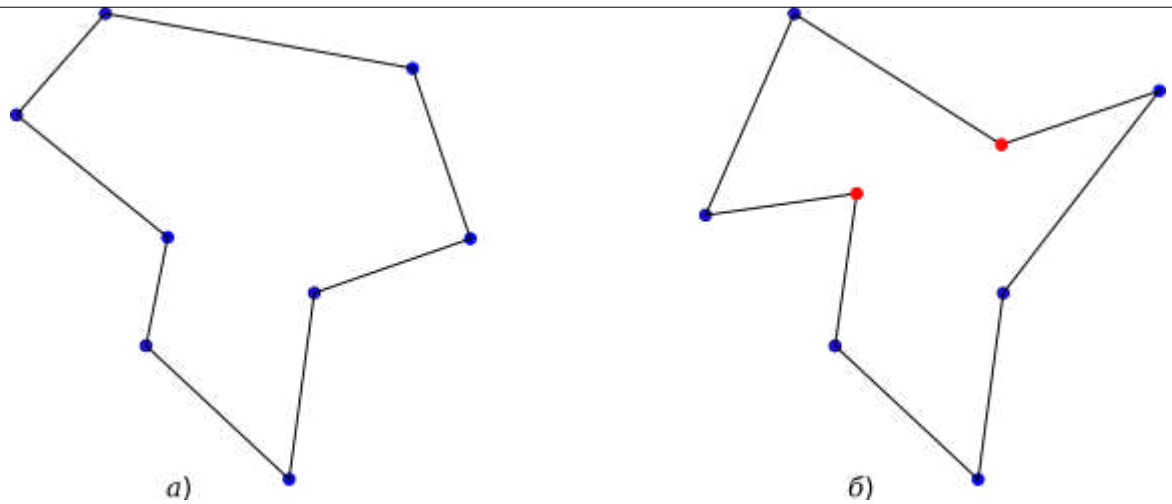


Рис. 4.2. Монотонний (а) та немонотонний (б) багатокутник

*Розбиття довільних простих багатокутників на монотонні частини*. Нехай $P$ є простим багатокутником і $v$ – деяка його вершина. Ребро $\{u, v\}$ є *верхнім* ребром відносно $v$, якщо $y$–координата $u$ більша за $y$–координату $v$. Ребро $\{w, v\}$ є *нижнім* ребром відносно $v$, якщо $y$–координата $w$ менша за $y$–координату $v$ (рис. 4.3а). Вершина $v$ є *регулярною*, якщо вона є найнижчою (рис. 4.3б) або найвищою (рис. 4.3в) вершиною $P$, або має і верхнє і нижнє ребро. Багатокутник $P$ є *регулярним*, якщо кожна його вершина є регулярною. Окрім нижньої та верхньої точки в ньому не може бути вершин з рис. 4.2б–в.

Лістинг 4.2 Регуляризація багатокутника

```
Функція ADD_UPPER_EDGES
Вхід: довільний багатокутник P
Вихід: багатокутник P', отриманий додаванням ребер до P таким чином, що кожна вершина
P' (окрім найвищої) має як мінімум одне верхнє ребро
1) Відсортуємо вершини в P за збільшенням y–координати
LET відсортований список L = {v(1), v(2), ..., v(n)}
2) Створюємо 2–3 дерево T
Заносимо верхні ребра v(1) в T, якщо вони існують, інакше позначаємо v(1) як "висячу"
вершину
3) FOR i = 2 .. n
        Знаходимо найближче до v(i) ліве e(1) і праве e(r) ребра в T з використанням
                                                                x–координати
        FOR j = 1 .. r–1
                IF існує висяча вершина v(h) між e(j) та e(j+1)
                        Додати нове ребро {v(h), v(i)}
                        Позначаємо v(h) як не висячу
        DELETE нижні ребра e(2), ..., e(r–1) вершини v(i) з T, якщо вони існують
        IF v(i) має верхні ребра
                Занести верхні ребра v(i) в T
        ELSE
                Розмістити v(i) як висячу вершину між найближчим лівим і правим ребрами
                                                        e(1) та e(r) IF i<>n
Функція ADD_LOWER_EDGES є симетричною
```

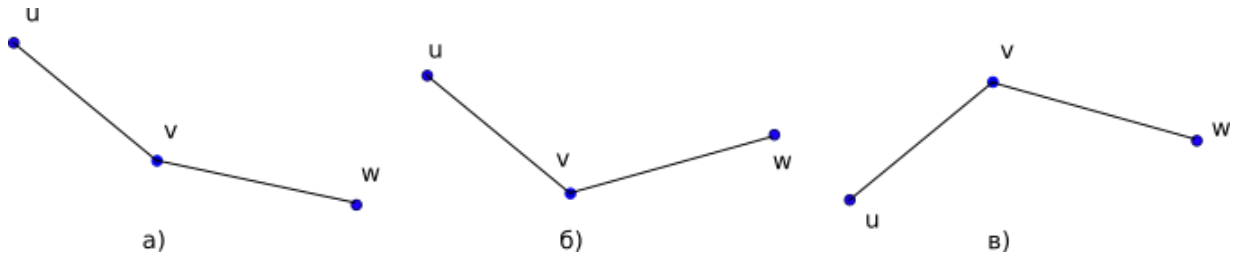


Рис. 4.3. Приклади регулярних вершин

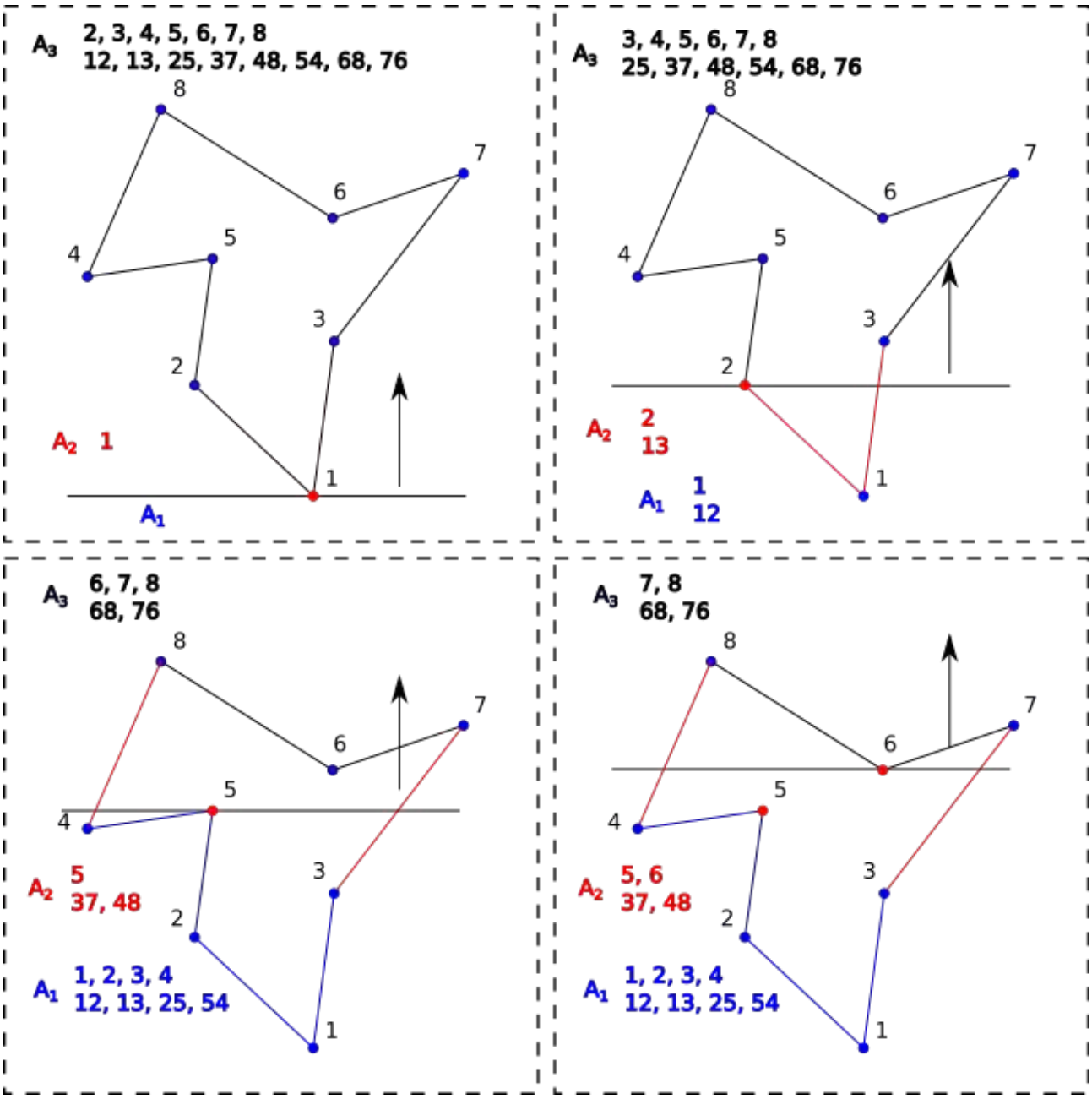


Рис. 4.4. Регуляризація багатокутника

Регуляризація багатокутника на монотонні частини методом розбиття на трапеції. Введемо додатково види нерегулярних вершин (для випадку замітання зверху–униз):

- merge–вершина – нерегулярна вершина з двома верхніми ребрами (рис. 4.3б);
- split–вершина – нерегулярна вершина з двома нижніми ребрами (рис. 4.3в).

Також введемо поняття *вершини–помічника helper(e)* для ребра *e* – найнижча вершина над замітаючою прямою така, що горизонтальний відрізок,

що поєднує *e* та *helper(e)* знаходиться цілком всередині багатокутника (рис. 4.5).

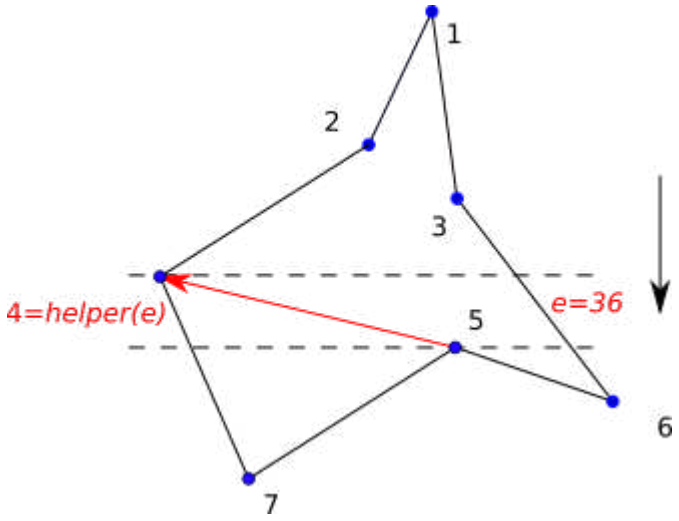


Рис. 4.5. Приклад вершини–помічника

Тоді для split–вершини *v*, у якої *e* – найближче ребро вздовж замітаючої прямої (зліва або справа):

- додати діагональ з *e* до *helper(e)* (рис. 4.5).

Для merge–вершини *v*, у якої *e* – найближче ребро вздовж замітаючої прямої (зліва або справа):

- *v* стає *helper(e)* в той момент, коли замітаюча пряма досягає *v*;
- в той момент, коли helper(e) буде замінено на деяку вершину $v_m$, додамо діагональ з $v_m$ до *v*;
- якщо *v* не буде замінено в якості *helper(e)*, то проводимо діагональ з *v* до нижньої вершини *e*.

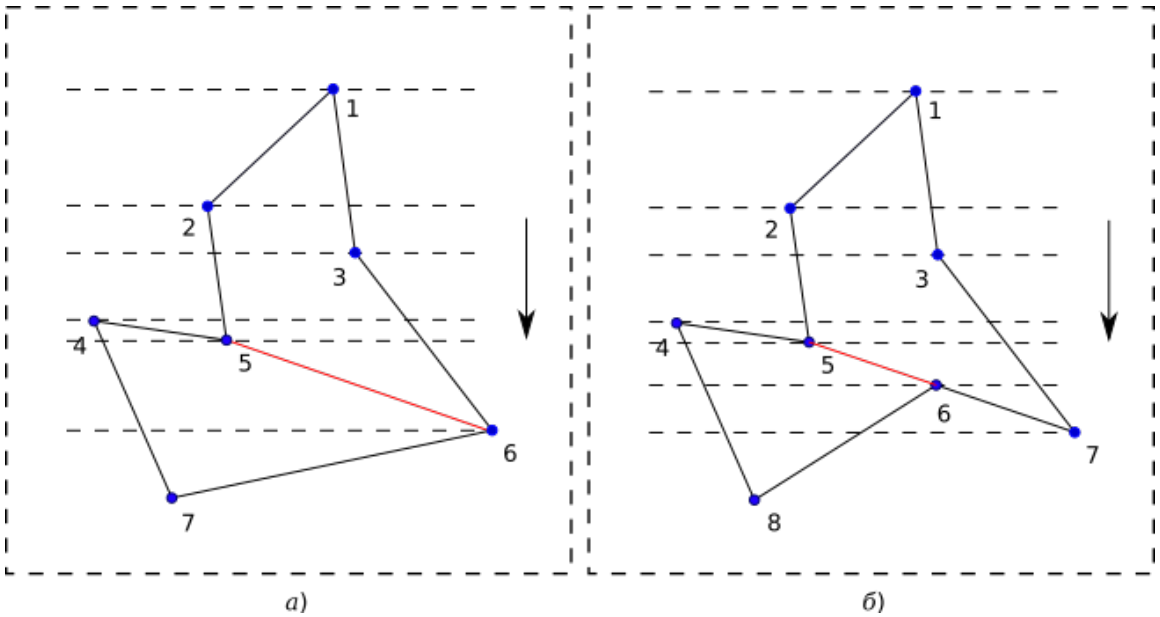


Рис. 4.6. Приклади регуляризації методом розбиття на трапеції

# 5 ДІАГРАМИ ВОРОНОГО

*Діаграма Вороного Vor(S)* множини $n$ точок $S=\{p_1, ..., p_n\}$ на площині – це таке розбиття площини на $n$ регіонів $V_1, V_2, ..., V_n$, що будь–яка точка в регіоні $V_i$ ближча до точки $p_i$, аніж до будь–якої іншої точки в множини $S$ (рис. 5.1).

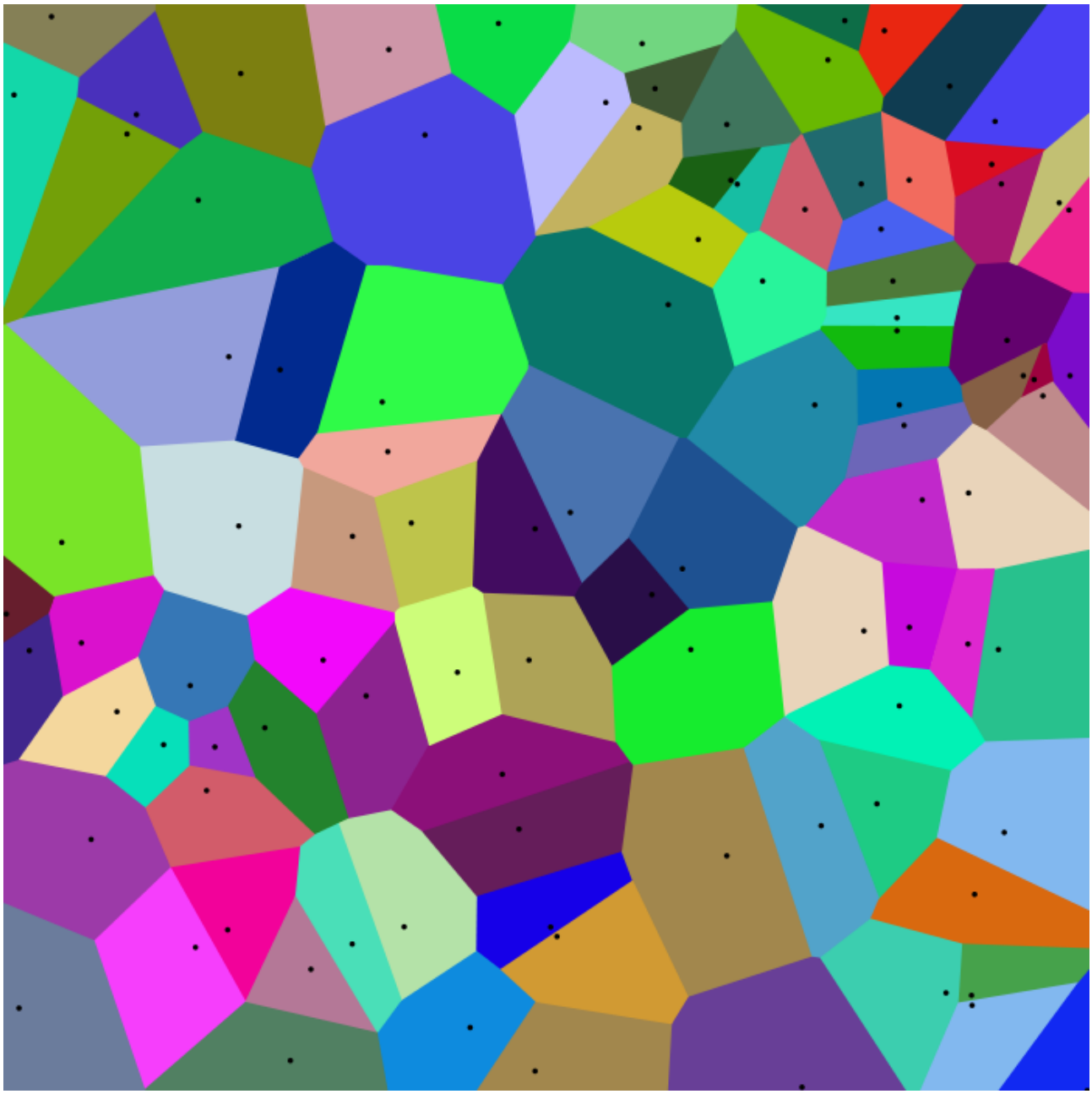

Рис. 5.1. Діаграма Вороного

Регіони $V_1, V_2, ..., V_n$ – *регіони Вороного*, вершини діаграми – *вершини Вороного*, а її ребра – *ребра Вороного*. Слід відмітити, що вершини Вороного в загальному випадку не є точками множини $S$.

*Припущення 5.1* Ніякі чотири точки із $S$ не лежать на колі.

*Твердження 5.2* Будь–яка вершина Вороного має ступінь 3.

*Твердження 5.3* Для будь–якої вершини Вороного $v$, що належить регіону $V_i$, коло радіуса $p_i v$, проведене з точки $p_i$, не містить всередині точок множини $S$.

*Твердження 5.4* Діаграма Вороного містить якнайбільше $2n-3$ вершин, $3n-5$ ребер та $n$ регіонів.

*Твердження 5.5* Регіон Вороного не має границі тоді і тільки тоді, якщо відповідна точка множини $S$ є вершиною опуклої оболонки $CH(S)$.

Розглянемо рекурсивний алгоритм побудови діаграми Вороного методом "розділяй і володарюй".

Лістинг 5.1 Рекурсивний алгоритм побудови діаграми Вороного

Вхід: множина S з n точками на площині
Вихід: діаграма Вороного Vor(S) множини S
1) Відсортуємо точки в S за x–координатами
2) CALL VORONOI(S)

Функція VORONOI(S)
Вхід: множина S з n точками на площині, відсортованими за x–координатами
Вихід: діаграма Вороного Vor(S) множини S
1) Розбиваємо S на дві підмножини SL та SR приблизно рівного розміру вертикальною прямою L таким чином, що усі точки в SL знаходяться по ліву сторону прямої L, а усі точки в SR – по праву сторону від L
2) CALL MERGEHULL(SL) для побудови діаграми Вороного Vor(SL)
CALL MERGEHULL(SR) для побудови діаграми Вороного Vor(SR)
3) CALL MERGE(Vor(SL), Vor(SR)) для отримання Vor(S)

Функція MERGE(Vor(SL), Vor(SR))
Вхід: діаграми Вороного Vor(SL) та Vor(SR)
Вихід: об'єднана діаграма Вороного Vor(SL+SR) множин SL та SR
1) CALL SIGMA(SL, SR) для побудови ламаної σ, що розділяє SL та SR
2) Видаляємо усі повні та неповні ребра Vor(SL), що цілком знаходяться по праву сторону від σ
3) Видаляємо усі повні та неповні ребра Vor(SR), що цілком знаходяться по ліву сторону від σ

Функція SIGMA(Vor(SL), Vor(SR))
Вхід: діаграми Вороного Vor(SL) та Vor(SR)
Вихід: ламана σ та опукла оболонка CH(S) множини S

1) Знаходимо CH(SL) та CH(SR)
2) Знаходимо верхній bU та нижній bL мости для CH(SL) та CH(SR)
3) Побудуємо серединні перпендикуляри lU та lL до мостів bU та bL, відповідно
4) Побудуємо CH(S) з допомогою мостів bU та bL
5) Обходимо ламану σ в напрямку зменшення y−координати починаючи з точки на кінці lU ламаної σ, що знаходиться в нескінченності (або достатньо вища за bU), будуючи σ ребро за ребром, доки не досягнуто lL
LET p0 – точка на lU, що достатньо вища за bU=(pL, pR), де точка pL знаходиться в SL, а точка pR – в SR.
LET l0 промінь з початком в p0, який має напрям, протилежний lU та VL і VR регіони Вороного для точок pL та pR в діаграмах Vor(SL) та Vor(SR), відповідно
WHILE l0 не співпадає з lL
Знаходимо точку qL, яка є точкою перетину l0 з границею VL, та точку qR, яка є точкою перетину l0 з границею VR
IF p0 ближча до qL, аніж до qR
Припустимо, що точка qL знаходиться на ребрі Вороного eL діаграми Vor(SL), яке задане точкою pL та деякою іншою точкою pL’ в SL, тоді p0=pL і l0 – промінь, що починається в точці qL, та є серединним перпендикуляром до відрізка {pL’, pR}, та проходить в напряму зменшення координати y
LET поточний регіон VL діаграми Vor(SL) є регіоном точки pL’
IF p0 ближча до qR, аніж до qL, то поновлюємо параметри p0, l0 та VR аналогічно

Сумарна часова складність алгоритму складатиме O($n \log n$).

# 6 ТРІАНГУЛЯЦІЯ ТОЧКОВИХ МНОЖИН. ТРІАНГУЛЯЦІЯ ДЕЛОНЕ

При розгляді багатокутників їх ребра чітко відокремлюються від діагоналей. Для точкової множини *S* поняття ребра використовується для позначення будь–якого відрізку, який має дві точки множини S в якості його кінцевих точок.

*Тріангуляція точкової множини S на площині* – розбиття площини, яке визначається максимальною кількістю неперетинних ребер, множина вершин яких є *S*.

Довільна точкова множина *S* з *n* точками має кількість тріангуляцій, що пропорційна числу Каталана від *n*.

Розглянемо простий алгоритм тріангуляції точкової множини *S*, точки якої знаходяться в *загальному положенні* – жодні три точки не належать одній прямій. Тоді тріангуляція буде складатись з таких кроків:

1. Знаходимо опуклу оболонку *CH(S)*.
2. Проводимо тріангуляцію *CH(S)* як багатокутника, ігноруючи усі внутрішні точки.
3. Внаслідок обмеження на загальне положення точок кожна внутрішня точка множини *S* знаходиться всередині деякого трикутника, а не на його ребрах. Тому, обираємо внутрішню точку і проводимо з неї ребра до вершин трикутника, в якому вона лежить.
4. Повторюємо крок (3) для кожної внутрішньої точки

На рис. 6.1 наведено три різні варіанти тріангуляції за представленим алгоритмом.

*Інкрементальний алгоритм тріангуляції.*

1. Відсортуємо точки *S* за x–координатою та помістимо їх в список *L*.
2. Перші три точки в *L* формують трикутник.
3. Розглянемо наступну точку *p* в *L* і з'єднаємо її з усіма попередніми точками, які видимі з *p*.
4. Повторюємо крок (3) для кожної точки з *L*.

На рис. 6.2 наведено приклад роботи інкрементальної тріангуляції.

*Види тріангуляції.* Внаслідок великої кількості можливих тріангуляцій для однієї точкової множини існує багато критеріїв, за якими може бути обрано той чи інший варіант тріангуляції. Серед важливих видів тріангуляції виділяють:

- *мінімальну зважену (оптимальну) тріангуляцію* – тріангуляція, в якій сумарна довжина усіх ребер мінімальна;
- *тріангуляцію Делоне* – для будь–якого трикутника в тріангуляції усі точки з $S$ за винятком його вершин, лежать поза колом, описаним навколо трикутника;
- *псевдотріангуляція* – розбиття площини на багатокутники, які містять три опуклі вершини.

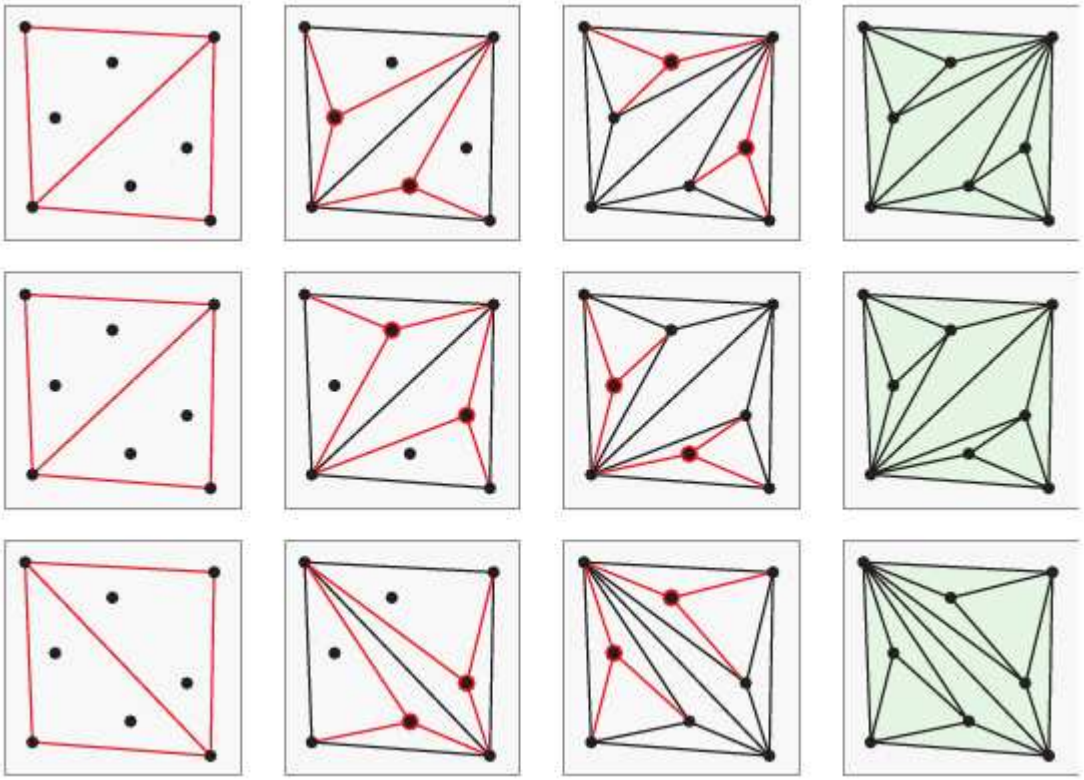

Рис. 6.1. Тріангуляція через побудову опуклої оболонки

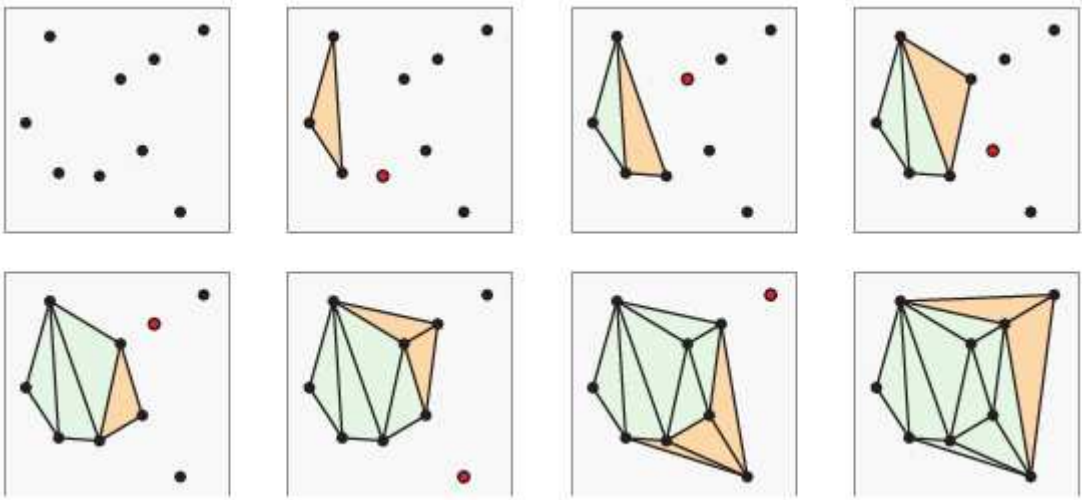

Рис. 6.2. Інкрементальна тріангуляція

Тріангуляція Делоне часто використовується в побудові карт висоти та візуалізації ландшафтів (рис. 6.3). Її відмінна риса – максимально “товсті” трикутники (рис. 6.4).

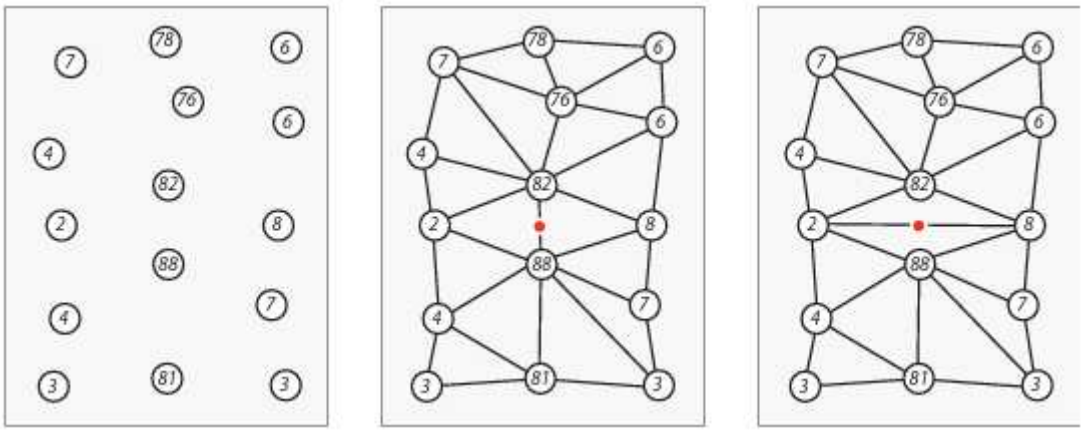

Рис. 6.3. Тріангуляція карти висот

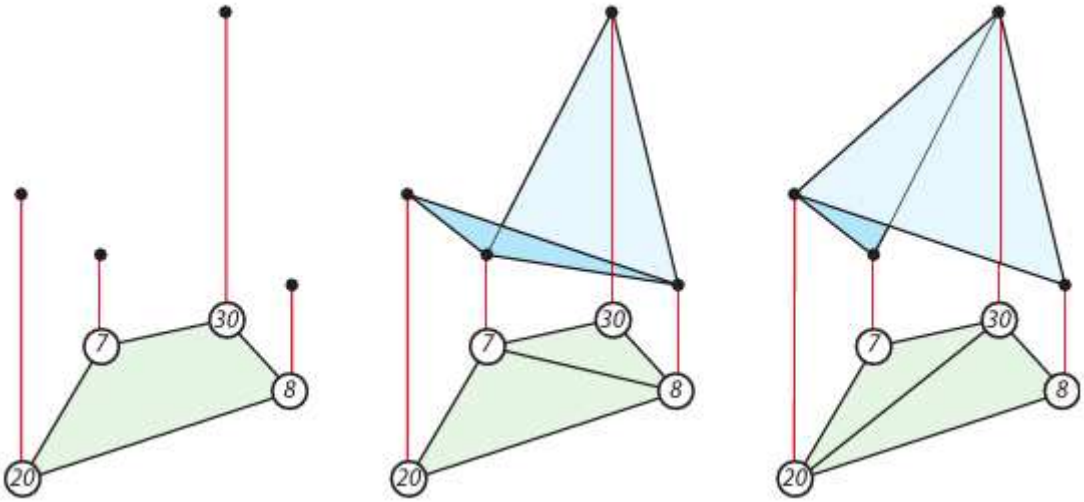

Рис. 6.4. Вплив тріангуляції на карту висот

Кожна точкова множина $S$ має тріангуляцію Делоне. Для того, щоб тріангуляція Делоне була єдиною для точкової множини $S$, необхідно щоб точки в $S$ знаходились в *загальному положенні* – жодні 4 точки не лежать на одному колі.

Введемо поняття *послідовності кутів* $(\alpha_1, \alpha_2, ..., \alpha_{3n})$ – список усіх $3n$ кутів для тріангуляції $T$ точкової множини $S$ з $n$ точками на площині, відсортований від найменшого кута $\alpha_1$ до найбільшого $\alpha_{3n}$.

Для двох тріангуляцій $T_1$ та $T_2$ множини $S$, $T_1$ є більш *товстою*, аніж $T_2$, (і позначатиметься $T_1 > T_2$) якщо послідовність кутів $T_1$ лексикографічно більша за таку в $T_2$.

Нехай $e$ – ребро в тріангуляції $T_1$ і $Q$ – чотирикутник в $T_1$, сформований двома трикутниками, які мають $e$ як загальне ребро. Якщо $Q$ є опуклим, то нехай $T_2$ буде тріангуляція після перевертання $e$ в $T_1$. Тоді $e$ буде *коректним* ребром, якщо $T_1 \geq T_2$ і *некоректним*, якщо $T_1 < T_2$.

Тріангуляція Делоне $Del(S)$ – тріангуляція, яка містить тільки коректні ребра.

Алгоритм тріангуляції Делоне:

1. Побудуємо будь–яку початкову тріангуляцію $T$.
2. Якщо в $T$ є некоректне ребро $e$, то робимо його коректним шляхом перевертання.
3. Повторюємо перевороти усіх некоректних ребер, доки не залишиться жодного.

На рис. 6.5 наведено графічну ілюстрацію визначення коректності ребер тріангуляції – для чотирикутника $ABCD$ ребро $AC$ є коректним, якщо точка $D$ лежить поза описаним колом для трикутника $ABC$, якщо точка $D$ лежить всередині описаного кола, то ребро $AC$ є некоректним. Випадок, коли точка $D$ належить описаному колу виникає за умови того, що точки множини не знаходяться в загальному положенні.

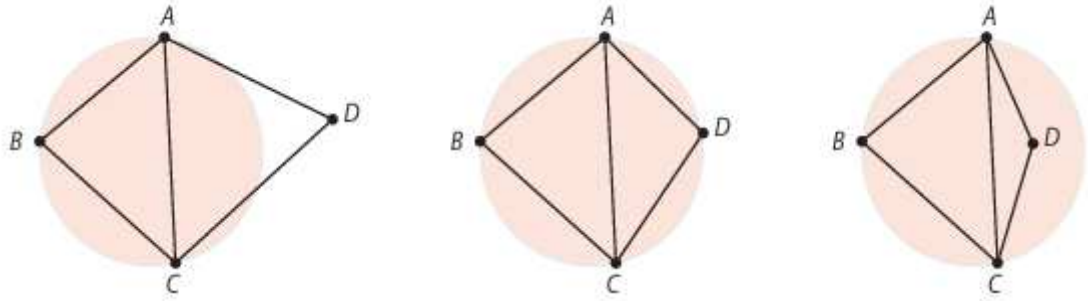


Рис. 6.5. Умова коректності тріангуляції Делоне

Тріангуляція Делоне представляє собою дуальний граф діаграми Вороного, що надає такий алгоритм тріангуляції:

1. Побудова діаграми Вороного для точкової множини.
2. Усі суміжні центри регіонів Вороного з'єднуємо ребрами, які формуватимуть коректну тріангуляцію Делоне.

# 7 ГЕОМЕТРИЧНИЙ ПОШУК. МЕТОДИ РОЗБИТТЯ ПРОСТОРУ. ПРОСТОРОВІ СТРУКТУРИ ДАНИХ

Однією з важливих задач, що виникає в комп'ютерній графіці, базах даних, геоінформаційних системах є геометричний пошук в просторових даних, що представлені в вигляді масивів чисел. Для оптимізації пошуку в таких даних використовують спеціальні структури даних та алгоритми роботи з ними. Наприклад, для пошуку в одномірних даних можливе використання збалансованих дерев пошуку, або 2–3 дерев (рис. 7.1), які зменшують час пошуку з O($n$) до O(log $n$). В такому дереві усі вершини містять деякі значення $v_i$, такі що в лівих піддеревах знаходяться значення менші або рівні до $v_i$, а в правому – більші за $v_i$. Це дозволяє проводити ефективний пошук за діапазоном, наприклад, для дерева, що зображене на рис. 7.1, можливою задачею (запитом) може бути така: знайти усі елементи, що знаходяться в діапазоні [18, 77].

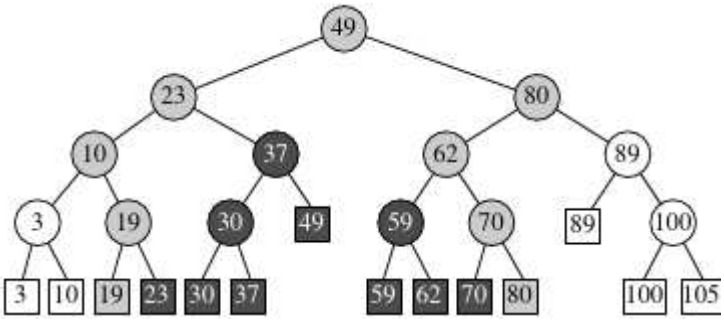


Рис. 7.1. Збалансоване бінарне дерево пошуку

Розглянемо алгоритм виконання пошукового запиту усіх елементів в збалансованому бінарному дереві пошуку.

Лістинг 7.1. Алгоритм пошуку в бінарному дереві

```
Вхід: бінарне дерево пошуку T, діапазон пошуку [x, x']
Вихід: усі точки в T, які лежать в заданому діапазоні
1) LET v_split – вершина в T, в якій розділяються шляхи до x та x'
v_split = CALL FIND_SPLIT_NODE(T, x, x')
2) IF v_split є листовою вершиною
        перевіряємо, чи належать точки в піддеревах v_split діапазону пошуку
ELSE //Проходимо по дереву до x та повертаємо усі значення в правих піддеревах шляху
        LET v = left(v_split) – ліве піддерево v_split
        WHILE v не є листовою
                LET x_v – значення, що зберігається в вершині v
                IF x ≤ x_v
                        LET right(v) – праве піддерево вершини v
```

CALL REPORT_SUBTREE(right(v))
LET v = left(v)
ELSE
LET v = right(v)
Перевіряємо, чи належить значення, що зберігається в v до діапазону
Подібним шляхом проходимо по дерево до x' та повертаємо усі значення в лівих піддеревах шляху та перевіряємо, чи належить точка в кінцевій листовій вершині шляху до діапазону
Функція FIND_SPLIT_NODE(T, x, x')
Вхід: бінарне дерево пошуку T, та значення x і x', такі що x ≤ x'
Вихід: вершина v, в якій розділяється шлях до x та x' або листова вершина, в якій обидва шляхи закінчуються
1) LET v = root(T) – корінь дерева
LET x_v – значення, що зберігається в вершині v
2) WHILE v не є листовою та x ≤ x_v або x > x_v
IF x ≤ x_v
v = left(v)
ELSE
v = right(v)
RETURN v

Функція REPORT_SUBTREE(subtree(v)) проводить пошук по піддереву
Вхід: піддерево subtree(v) вершини v
Вихід: значення усіх листових вершин в заданому піддереві

*k–мірні дерева пошуку (kd–дерева)* – узагальнення бінарних дерев пошуку на простори довільної розмірності, представляють собою бінарні дерева, в яких кожен внутрішній вузол містить одну точку та відповідає паралелепіпеду, який визначається цією точкою (рис. 7.2, 7.3). Паралелепіпед представляє собою опуклу множину, яку обмежують опуклі багатокутники з попарно паралельними опорними гіперплощинами. Кореневий вузол відповідає повному паралелепіпеду, що відповідає області інтересу. Кожен паралелепіпед розділяється на дві частини гіперплощинами, ортогональними до однієї з координатних осей, при цьому на кожному рівні дерева координатні осі циклічно змінюються.

Нові точки вставляються в дерево шляхом обходу його в глибину до листової вершини. Паралелепіпед, в який буде вставлено нову точку поділяється на дві частини вздовж координатної осі, яка відповідає поточному рівню дерева.

Основна проблема при використанні *kd*–дерев – залежність їх форми від порядку, в якому додаються точки. В найгіршому випадку для $n$ точок потрібно $n$ рівнів дерева, що приводить до лінійного часу пошуку в дереві O($n$). Для вирішення цієї проблеми запропоновано адаптивні kd–дерева, в яких множини точок на кожному рівні поділяються на приблизно рівні підмножини (рис. 7.4). Дерева, побудовані таким способом є збалансованими і будь–який пошук в них виконуватиметься за O(log $n$), а побудова повного дерева – O($n$ log $n$).

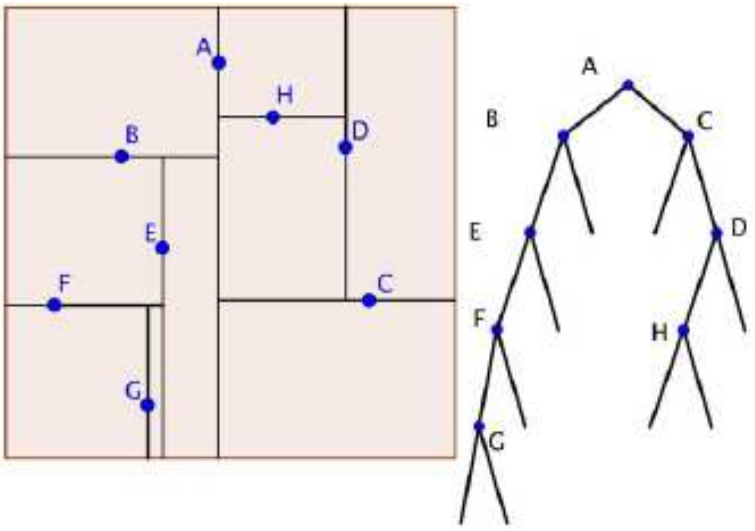


Рис. 7.2. Двовимірне дерево (k=2)

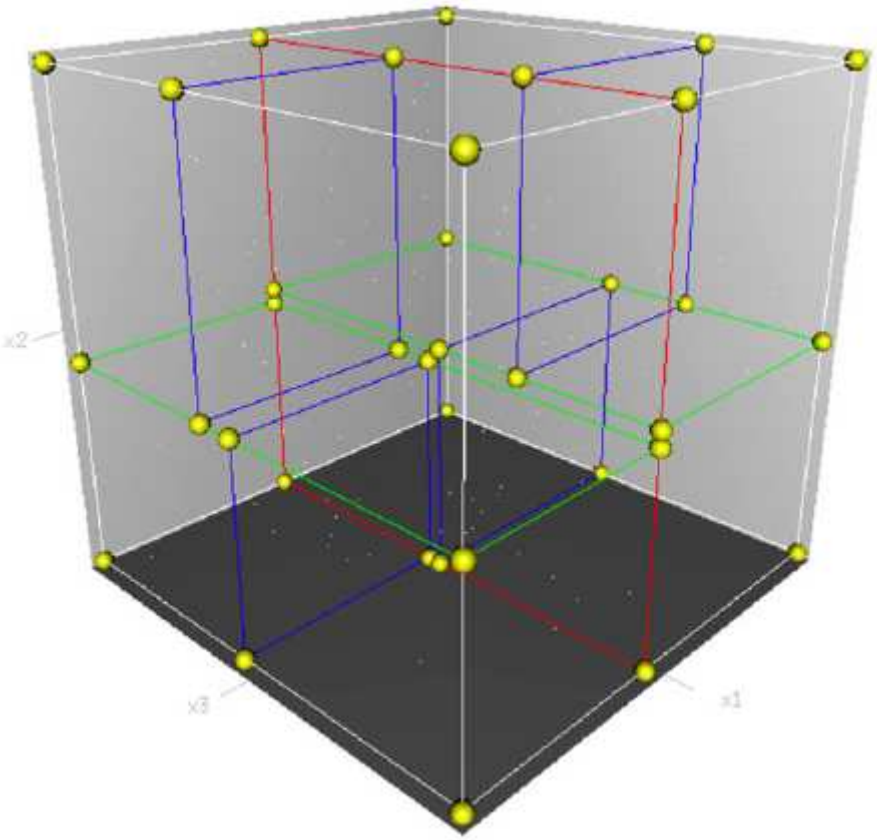

Рис. 7.3. Тривимірне дерево (k=3)

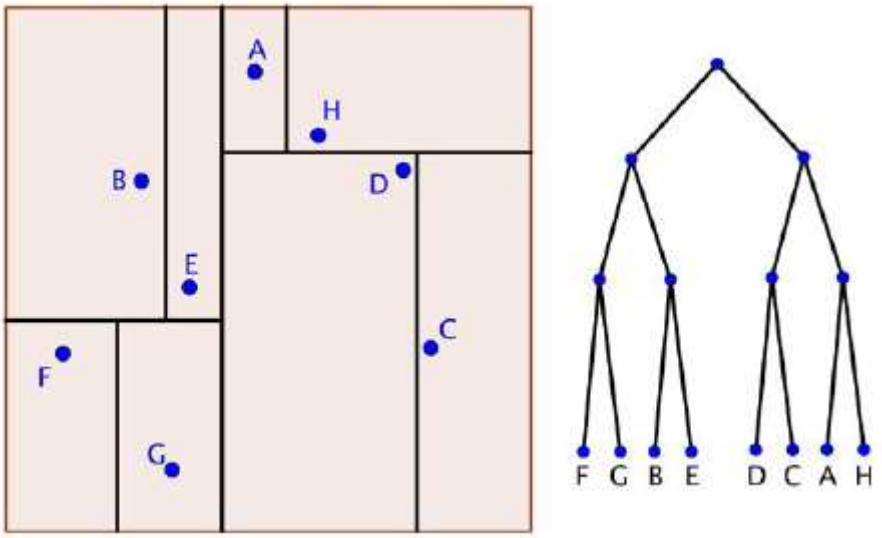


Рис. 7.4 Адаптивне kd–дерево

*BSP–дерева (binary space partitioning)* – представляють бінарне розбиття простору довільними напівплощинами (рис. 7.5). Найбільше підходять для зберігання напівплощин, відрізків (рис. 7.6) та багатокутників. Слід відмітити, що *kd*–дерева є спеціальним випадком *BSP*–дерев, в яких напівплощини перпендикулярні до однієї з координатних осей.

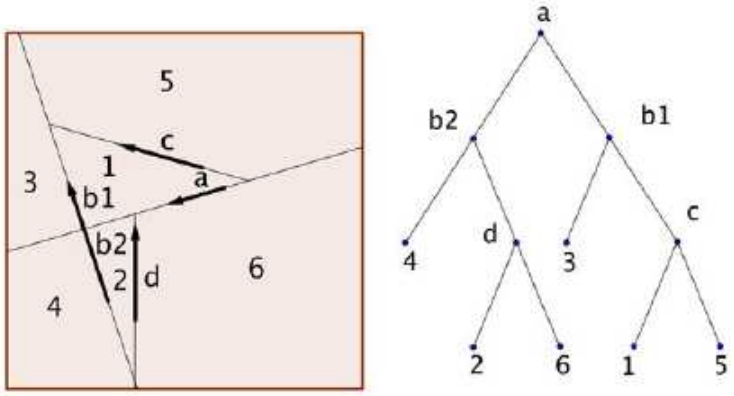


Рис. 7.5. *BSP*–дерево з напіплощин

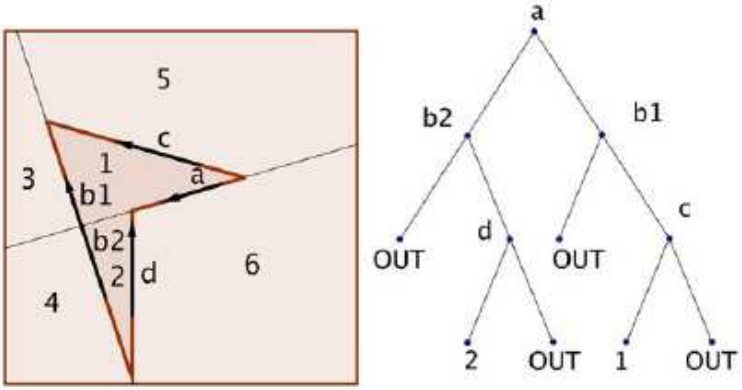


Рис. 7.6. *BSP*–дерево з відрізків

*Квадродерева* – узагальнене поняття для усіх видів дерев, що отримані шляхом рекурсивного розбиття двовимірного простору на чотири квадранти. Кожен вузол квадродерева завжди містить чотири нащадки (*квадранти*), які позначаються SW (південно–західний), NW (північно–західний), NE (північно–східний) та SE (південно–східний), або нумеруються числами {0, 1, 2, 3}, відповідно і зберігаються на кожному рівні в такому порядку. Усі вузли на одному рівні дерева мають однакову площу та розмір. На кожному рівні квадрант розбивається на чотири нові квадранти але необов'язково в порядку слідування вхідних точок, тому квадродерева можуть не бути збалансованими і, відповідно, потребуватимуть лінійного часу в найгіршому випадку.

Найчастіше використовують такі види квадродерев:

- точкове квадродерево (рис. 7.7)
- точка–регіон квадродерево (рис. 7.8)
- квадродерево регіонів (рис. 7.9)

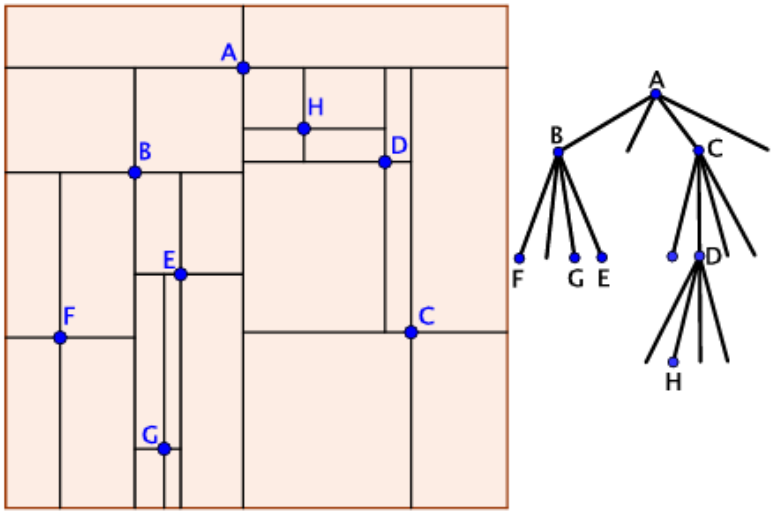


Рис. 7.7. Точкове квадродерево

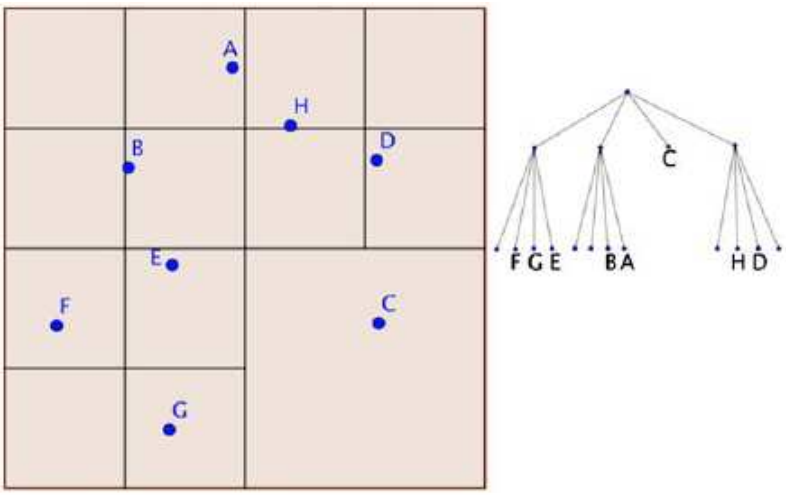


Рис. 7.8. Точка–регіон квадродерево

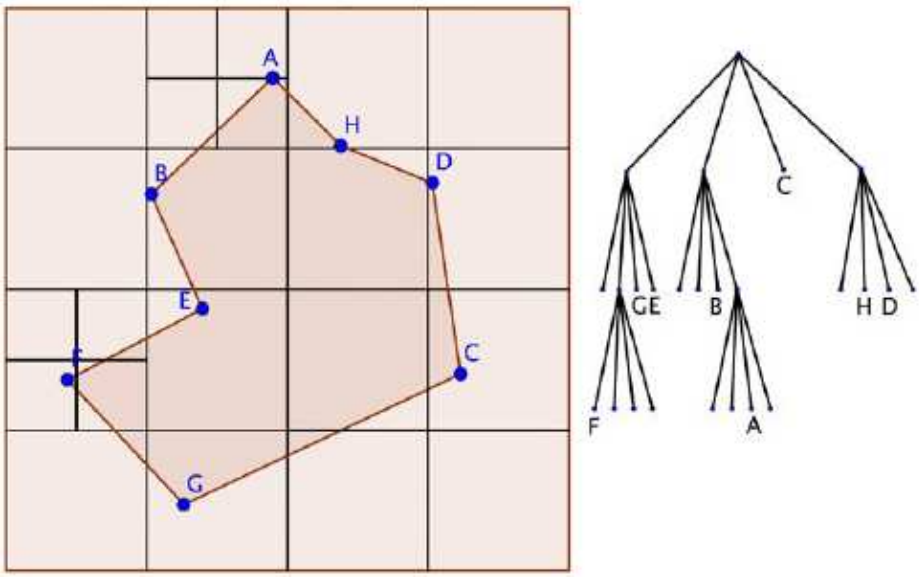


Рис. 7.9. Квадродерево регіонів

В точкових квадродеревах квадранти визначаються вхідними точками, горизонтальні та вертикальні прямі проходять через вхідні точки. В квадродеревах точка–регіон квадранти не проходять через точки, але на кожному рівні один з чотирьох квадрантів, якнайменше, містить кожну вхідну точку.

В квадродереві регіонів існують два типи листових вузлів: білі листи, які не покриваються відрізками, що обмежують вхідних регіонів, та чорні листи, які покриваються відрізками, відповідно.

*Октодерева* – є аналогами квадродеревах в тривимірному просторі (рис. 7.10). Представляють собою 8–арні дерева, в яких кожен нелистовий вузол має вісім нащадків, які мають назву *октанти*. Алгоритми роботи подібні до таких для квадродерев.

Також часто використовуються R–дерева, які групують вхідні об’єкти в ієрархічну структуру з використанням мінімальних прямокутників, що обмежують, або їх *n*–вимірних аналогів – мінімальних гіпер–прямокутників, що обмежують (рис. 7.11).

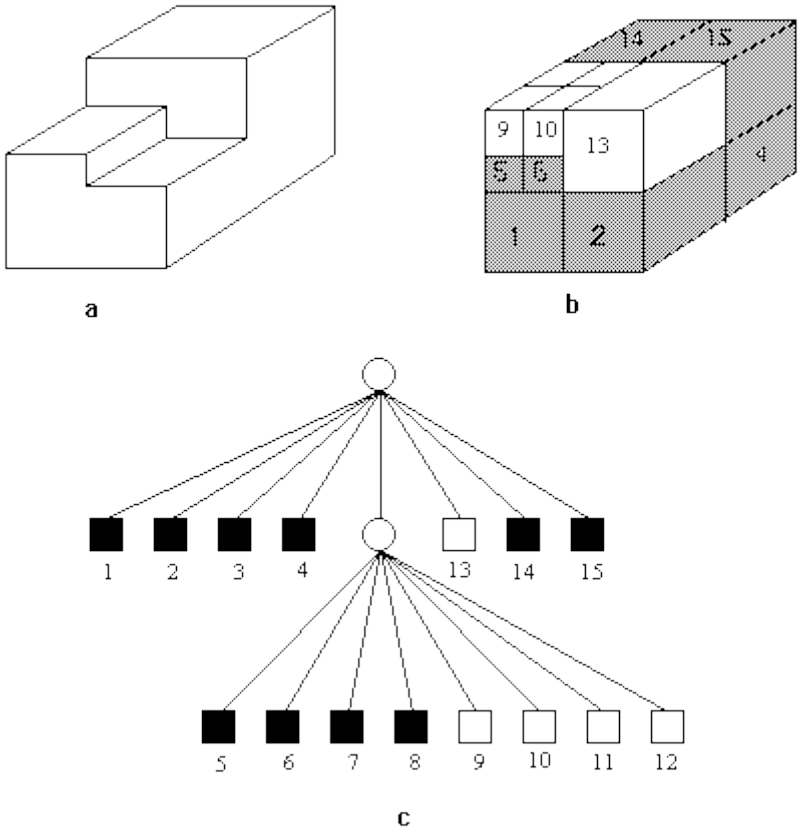


Рис. 7.10. Октодерево

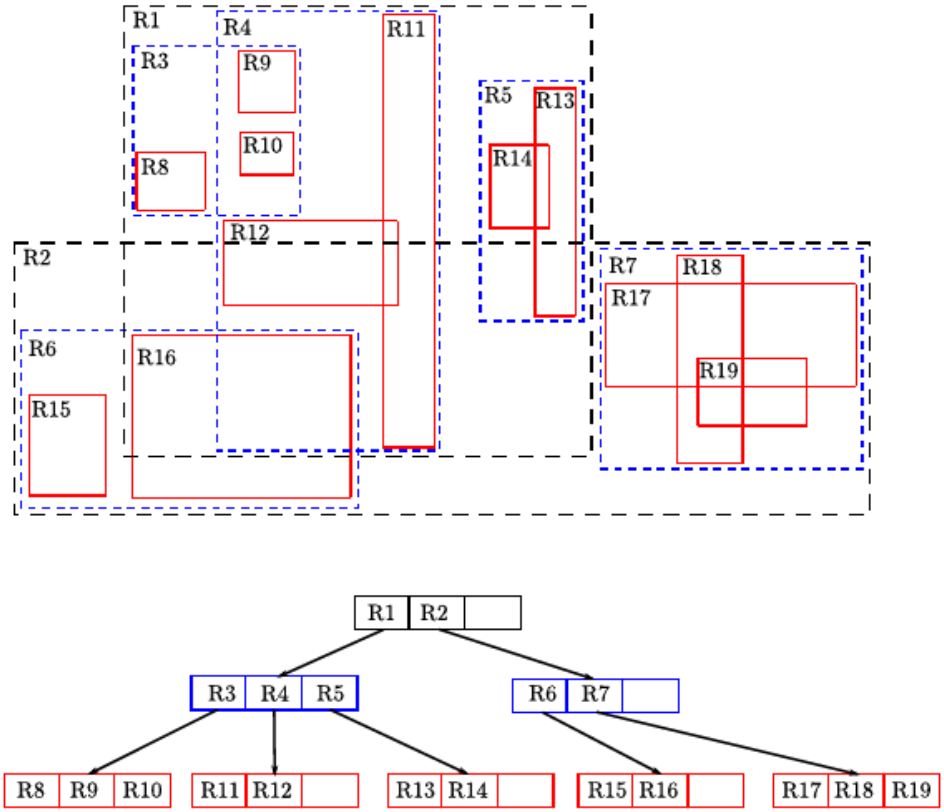


Рис. 7.11. R–дерево

# 8 ПОШУК НАЙБЛИЖЧИХ СУСІДІВ. ПРОСТОРОВЕ ХЕШУВАННЯ

В задачах комп'ютерної графіки та машинного навчання часто виникає задача пошуку найближчих сусідів до заданої точки, до таких задач можна віднести, наприклад, тріангуляцію, відстежування зіткнень об’єктів, задачі класифікації та кластеризації даних, апроксимація та виділення поверхонь з точкових даних. Постановка задачі пошуку найближчих сусідів: для заданої множини точок $S$, метричного простору $M$ та точки $q \in M$ знайти найближчу до $q$ точку в $S$. Узагальненням цієї задачі є задача пошуку $k$–найближчих сусідів ($k–NN$), в якій необхідно знайти $k$ точок, найближчих до $q$.

Найпростішим способом пошуку є лінійний перебір відстаней від точок $S$ до точки $q$, який має часову складність $O(d\ n)$, де $d$ – розмірність $M$. В лінійному пошуку не використовуються структури даних для підтримки роботи, тому в ньому відсутня просторова складність окрім зберігання точкової множини. Інший підхід базується на методах розбиття простору, в рамках цього підходу для розв’язання задачі пошуку найближчих сусідів можуть використовуватись такі структури даних, як $kd$–дерева, $BSP$–дерева, $R$–дерева, але ефективність цих структур даних падає з ростом розмірності даних. Це призвело до появи алгоритмів приблизного пошуку найближчих сусідів, до яких можна віднести:

- методи жадного пошуку в графах близькості або графах відносного сусідства;
- методи просторового хешування – locality sensitive hashing (LSH) та ін.;
- пошук в просторах меншої розмірності – дерева покриттів;
- проективний радіальний пошук – для тривимірних щільних точкових множин, полягає в проектуванні точок на двовимірну сітку;
- методи на основі вектору ознак;
- методи зниження розмірності та кластеризації.

*Просторове хешування*. Для вирішення проблеми розмірності при пошуку найближчих сусідів може використовуватись метод просторового хешування, який полягає у відображенні багатовимірних даних в одновимірні значення (рис. 8.1). Загальна схема просторового хешування:

- простір $M$ розбивається на сітку з розміром $c$;
- для кожної клітини сітки розраховується її унікальний номер – *індекс*;

- для кожної точки множини $S$ розраховується значення *хешу* $h$, усі точки, що знаходяться в клітині сітки з індексом $i$ матимуть однакове значення $h$.

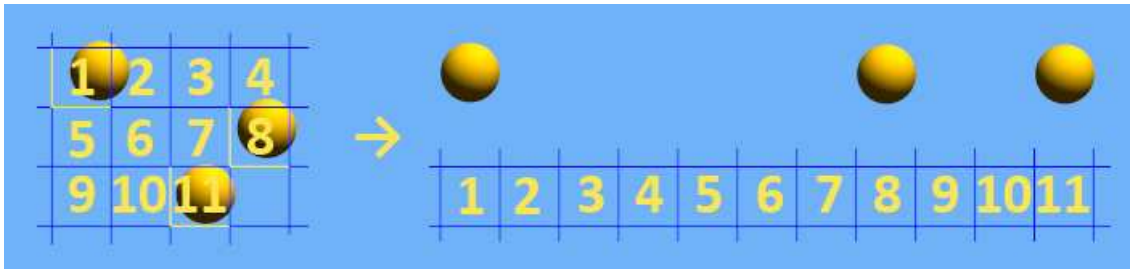


Рис. 8.1 Ілюстрація просторового хешування

Вибір функції розрахунку хешу проводиться таким чином, щоб точки, більш близькі в просторі $M$, мали близькі значення хешу:

$|A - B| < |B - C| => |h(A) - h(B)| < |h(B) - h(C)|$,

де $A$, $B$, $C$ – точки, $|\cdot|$ – функція визначення відстані між точками (*метрика*).

В комп'ютерних іграх найчастіше використовують таку функцію хешування:

$$h = \frac{x}{c} \cdot 2^k + \frac{y}{c} \cdot 2^m + \frac{z}{c} \cdot 2^n,$$

де $c$ – розмір сітки;

$k$, $m$ і $n$ – деякі константи, такі що, $k > m > n$ (або $k < m < n$).

Інша функція, яка запропонована для задач рендерингу:

$h = ((x \cdot p_1)$ XOR $(y \cdot p_2)$ XOR $(z \cdot p_3))$ MOD $n$,

де $p_1 = 73856093$, $p_2 = 19349663$, $p_3 = 83492791$ – великі прості числа,

XOR – побітова операція "Виключне АБО",

MOD $n$ – операція знаходження залишку від ділення на $n$,

$n$ – загальна кількість точок.

Однією з проблем просторового хешування є можливість колізій хешів – точки в різних областях простору можуть отримати однакові значення хешів. Для вирішення проблеми колізій запропонована проста функція просторового хешування для двовимірних точкових множин:

1) простір $M$ розбивається на сітку з розміром $c$;

2) для координат усіх точок $(x, y)$ розраховуються індекси ($j$ – номер колонки, $i$ – номер рядку в сітці) клітини, в якій знаходиться точка:

$$j=\frac{x}{c},$$

$$i=\frac{y}{c},$$

3) за отриманими індексами розраховуються значення хешу:

$h=j\bullet 10^{n}+i\bullet 10^{m}$,

де $n$ і $m$ – невеликі цілі числа, значення яких залежать від кількості розрядів в $j$ и $i$, відповідно (в загальному випадку $n=m+1$, $m=0$).

Узагальнений варіант хеш–функції для довільної розмірності даних:

1) простір розбивається на сітку розміром $c$;

2) для кожної точки с координатами ($x_1$, ..., $x_i$, …, $x_N$), що задані на діапазоні [$0$, $G_i$] розраховуються індекси клітини, в якій знаходиться ця точка:

$$i_i=\frac{x_1}{c},$$

...

$$i_N=\frac{x_N}{c},$$

де $N$– розмірність простору;

3) за отриманими індексами розраховуються значення хешу:

$$h_k=\frac{i_i\bullet 10^{(N-i)\bullet d}}{N},$$

$d$ – максимальна кількість розрядів хешу на одну просторову координату $x_i$,

$$i_i=max\left(\frac{G_i}{c}\right).$$

4) алгоритм розбиття отриманого хешу на окремі розряди:

```
t = N
WHILE t > 0
        i_i = h mod 10^d
        h = h / 10^d
        t = t – 1
```

## СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Препарата Ф., Шеймос М. Вычислительная геометрия: введение [Текст] – М.: Мир, 1989. – 478 с.

2. Никулин Е.А. Компьютерная геометрия и алгоритмы машинной графики [Текст] – Санкт-Петербург: БХВ-Петербург, 2003. – 560 с.

3. de Berg M., Cheong O., van Kreveld M., Overmars M. Computational geometry. Algorithms and applications [Текст] – Berlin: Springer, 2008. – 386 p.

Навчальне видання

**Обчислювальна геометрія в задачах комп'ютерної графіки та комп'ютерного зору**

Конспект лекцій для студентів спеціальності 122 Комп'ютерні науки

Укладач ДАШКЕВИЧ Андрій Олександрович

Роботу до видання рекомендувала проф. Шоман О.В.

В авторській редакції

План 2018 р., поз. 364.
Підписано до друку 15.03.2019 р. Гарнітура Таймс. Ум. друк. арк. 2,9.

---

Видавничий центр НТУ "ХПІ".
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

---

Самостійне електронне видання